上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

THESIS OF BACHELOR

论文题目：针对可扩展内存完整性保护的设计与实现

学生姓名：冯二虎

学生学号：516030910082

专　　业：软件工程

指导教师：夏虞斌教授

学院(系)：电子信息与电气工程专业

针对可扩展内存完整性保护的设计与实现

摘　要

内存保护是计算机安全中的重要组成部分，针对内存的攻击被划分为软件攻击与硬件攻击，软件攻击可以通过内存隔离防御，而硬件攻击需要通过内存加密与完整性保护来防御。内存隔离和加密在计算机系统与安全中都有较为成熟的解决方式，但是内存完整性保护还面临着内存大小与性能之间的权衡。2016 英特尔推出了 Software Guard Extensions（SGX），SGX 能够实现内存隔离，加密与完整性保护，从而为 enclave 程序提供了可信执行环境。虽然 SGX 提供了强安全保障机制，但是 enclave 能够使用的内存仅仅只有 128M，受限于内存的完整性保护。

在本文中，提出了一种新颖的可扩展内存完整性保护方案：挂载式完整性保护树。它能够保护 TB 级别的内存数据，实现离散的内存保护与动态的安全内存分配。同时，我们为完整性保护树提供了动态原语：挂载与卸载，以代替 SGX 中采用的页交换机制，实现了可扩展的内存保护。

我们在 gem5 上实现了可挂载完整性保护树的原型，并且在 SPECCPU 上做了初步的测试。测试的结果表明，采用了可挂载完整性保护树相较于之前先进的 VAULT 和 SIT 提升了 2~3 倍的性能，在极限的场景下能够提升 7~8 倍的性能。可挂载完整性保护树为可信执行环境，非易失性内存等新兴的应用场景与硬件提供了强安全保障，弥补了内存保护中，完整性保护的缺陷。

关键词：完整性保护，内存控制器，计算机安全

SCALABLE MEMORY INTEGRITY PROTECTION DESIGN AND RESEARCH

ABSTRACT

Memory protection is an important part of computer security. Attacks against memory are divided into software attacks and hardware attacks. Software attacks can be defended by memory isolation, while hardware attacks need to be defended by memory encryption and integrity protection.

Memory isolation and encryption have relatively mature solutions in computer systems and security, however memory integrity protection still face the trade-off between memory size and performance. In 2016, Intel launched the Software Guard Extensions (SGX), SGX enables memory isolation, encryption, and integrity protection, which provides a trusted execution environment for enclave. However although SGX provides strong security mechanism, enclave can only use 128M for memory, which is limited by memory integrity protection.

In this paper, a novel scalable memory integrity protection scheme is proposed: Mountable Merkle tree (MMT), which can protect TB level memory data and realize discrete memory protection and dynamic secure memory allocation. At the same time, we provide dynamic primitives for integrity protection trees: mount and unmount, which can significantly improve the performance against the page swapping mechanism adopted in SGX. We implement a prototype of a mountable merkle tree on gem5 and do the preliminary testing on SPECCPU. The results of the tests show that using the mountable merkle tree can improve 2 to 3 times better performance compared to VAULT and SIT, and 7 to 8 times performance improvement in the extreme scenarios. Mount Merkle tree can provide robust memory security protection for the trusted execution environment, non-volatile memory and other emerging scenarios and new hardware, and make up the integrity protection defects in traditional memory protection.

Key words: integrity protection, memory controller, computer security

目 录

插图索引

表格索引

第一章 概述

1.1 研究背景

在数字化的当下，计算机安全越来越受到厂商与消费者们的关注，日益复杂的软件程序和各异的硬件平台暴露出了更多的安全隐患，其中与内存相关的攻击占了绝大多数。基于内存的攻击可以分为两类：一类是软件攻击；另一类是硬件攻击。近年来研究者们通过新的硬件特性与软件架构，提出了诸多解决方案，其中内存完整性保护是不可或缺的一点。内存的完整性保护机制保证了内存不会受到诸如修改、拼接、重放等物理攻击，提供了很强的安全保障。传统的完整性保护使用的是 merkle tree，通过树状结构，父节点中保存子节点的 hash 值，最后所有的数据将受到 root hash 的保护。merkle hash 发明以来受到广泛的应用，在文件系统、网络传输中起到了至关重要的作用近几年，BMT，SIT 等新的数据结构相继提出，尝试解决 merkle tree 存在的缺陷。

2016 年，英特尔提出了 Software Guard Extension（SGX），是第一个商用的可信执行环境并且提供了内存隔离，内存加密与内存完整性保护等强内存保护机制。SGX 一经推出就受到了学术界和工业界广泛的使用。虽然 SGX 提供了很强的安全保障，但是支持的内存只有 128/256M，无法满足云场景下的需求，而限制 SGX 所保护内存大小的一个重要因素就是内存完整性保护机制。在 SGX 中内存完整性保护的数据结构为 Sgx Integrity Tree（SIT）。随着保护的内存增加，SIT 的深度也随之增加，随之将带来严重的运行开销。SGX 通过限制 SIT 的深度（即限制了所保护内存的大小），来确保运行时的开销在一个可接受的范围内。

在云计算普及的当下，计算机对内存的需求也越来越大，传统的 merkle tree 或者其变种无法支持在云场景下的 TB 级别的内存保护，2018 年学术界提出了 vault，用于扩展 SIT 所保护的内存空间。虽然 vault 将受保护的内存扩大到了 GB 的量级，但是仍然无法满足云场景下的需求。另外所有的内存完整性保护方案都不支持动态的内存完整性保护，需要预留 15-25% 总内存用于储存完整性保护树的数据结构，这样的数据结构在云场景中与可扩展与弹性的内存分配机制相违背。为了解决内存完整性保护大小以及可扩展性问题，使内存完整性保护方案能够适用于云端，为云端计算提供更强的内存保护机制，本文提出了一种新型的内存完整性保护数据结构，尝试解决这两个内存完整性保护工作中的痛点。通过将静态完整性保护树演变为动态可挂载的完整性保护树，给予了内存完整性保护更多的弹性与可扩展性。

1.2 研究现状

本节将围绕内存完整性保护数据结构以及应用场景做进一步介绍

1.2.1 内存完整性数据结构

Merkle hash tree 是使用最广泛的完整性保护的数据结构，之后的很多工作都是基于此做进一步的优化。完整性保护的核心想法是用较小的 hash 保护大量的数据内容不被修改，显然仅通过一次 hash 操作来实现数据的完整性保护是不安全，因为只有一个 hash 值将极大的增加 hash 碰撞的概率，同时需要扫描完所有的数据之后，才能够计算出 hash。在 Merkle hash tree 中采用了多个 hash 值的方式保护数据的完整性，同时为了减少可信的 hash 数量，

作者采用了树状的结构来管理所有的 hash 值。首先先将数据进行分块，每个数据块大小相同。其次对每个数据进行 hash 计算，得到对应的 hash 值。然后将连续地 N 个 hash 值作为新的数据块，在此基础上再进行一次 hash 计算，得到新的 hash 值。以此迭代，直到得到唯一地 hash 值为止。需要对数据进行完整性检查的用户只需要持有根节点中的 hash 值，然后通过 hash tree 的算法方式，通过数据重新计算出根节点中的 hash，并且将保存的 hash 值与计算出的 hash 值进行比较，如果两者一直就说明数据的完整性得到保证。

Merkle tree 的提出，使得完整性保护得计算不需要遍历所有得数据，极大得提高了完整性保护效率。同时只需要保存根节点得 hash 值就能保护所有得数据得完整性，减少了保存可信 hash 值空间开销。得益于 Merkle tree 得精妙得设计，Merkle tree 再数据完整性保护领域中得到了广泛得使用，之后所有的数据完整性结构也都是 Merkle tree 得变体，虽然 Merkle tree 设计非常成功，但是也存在诸多缺陷，在数据越来越多的当下，Merkle tree 深度逐渐增加，运行时的开销也逐渐增加；从而进一步演化出了基于计数的完整性保护方案，其中代表就是 Bonsai Merkle Trees。通过引入计数，增加了树的扇出，从而再相同的层数下，保护的内存更多。同时，基于计数的完整性保护方案也能够很好的和内存加密结合再一起，提供完整性和机密的双重保护。

1.2.2 可信执行环境-Enclave

enclave 是计算机中中一块特殊的执行环境，在 enclave 中运行的代码无法被 OS 和 hypervisor 访问与管理，同时对于内存中的数据都是加密的形式保存，并且能够提供完整性的保护。基于 enclave 提供的内存隔离，内存完整性保护，内存加密，能够使运行在 enclave 中程序免受软件（例如恶意的 OS 以及 hyperbisor）与硬件的攻击（冷冻内存）。2016 年，Intel 公司发布了第一代 sgx，sgx 是第一个商用的 enclave 架构，通过微码实现在 Intel 的 6 代即以上的 CPU 中，在提供极高的安全保障（内存隔离，内存完整性保护，内存加密）。同时 sgx 的可信基被限制在 cpu 的内部，任何外部硬件，总线，以及任何特权软件都在可信基之外。正是因为 sgx 提供了如此强的安全机制，在云场景中得到了广泛的使用，尤其是正对云上的可信计算，加解密等与安全密切相关的应用。在 intel 之后，amd、arm、riscv 等不同计算机指令集与制造上分别提出了各自的 enclave 架构。但是并没有提供和 sgx 一致的安全等级，指实现了内存隔离或加密，没有提供内存完整性保护。而其中最主要的因素就是已有的内存完整性保护机制无法很好的保护较大的内存。

1.2.3 非易失内存-NVM

非易失性内存是新的存储介质，它同时兼备内存和硬盘的特性。非易失性内存像内存一样，可以直接插在 DIMM 上，并且 CPU 可以根据 byte 进行寻址；同时非易失内存于普通内存最大的区别在于，断电之后数据不会丢失，同时相较于普通的内存，非易失内存可以做到更大的容量，一根非易失内存就可以达轻松达到 512GB，1TB 的大小。在性能方面，非易失内存在读写方面和内存相近，但是存在明显的读写不平衡，跟固态硬盘相似非易失内存也存在写穿的问题。得益于非易失性内存的诸多好处，学术界和工业界对非易事性内存做了广泛的研究，其中如何保证非易失性内存中数据的安全变得格外的重要。与普通的内存不同，非易失性内存中的数据在断电之后并不会丢失，所以攻击者可以非常方便的将非易失内存从 DIMM 上拔下来，然后插到另外的主机上读取其中的数据。非易失内存的出现，使得针对内存的攻击更加可行，所以对内存的完整性保护也更为主要。同时由于非易失性

内存的大小往往在 TB 左右，导致了已有的完整性保护方案并不能直接迁移到非易失性内存中，同时写穿等问题也会带来额外的挑战。

1.3 主要研究内容

本文首先介绍了不同的完整性保护方案，数据结构以及他们各自优缺点。然后结合内存完整性保护的需求与场景，指出已有的完整性保护的方案都不能很好适应云场景。本文具体分析了其中的原因，第一：完整性保护的内存大小受限；第二：无法动态的分配受保护的内存；第三：无法支持离散的内存保护。本文针对已有完整性保护方案中的这三点缺陷，提出了一种新的完整性保护方案：可挂载的完整性保护树（Mountable merkle tree），并且提出了两个完整性保护树的动态原语，挂载（Mount）与卸载（unMount）。可挂载的完整性保护树能够很好的和兼容已有的完整性保护方案，在不需要更改已有数据结构基础上，增加少量元数据结构以及硬件扩展，就能解决原有的完整性保护方案的三个主要缺陷。同时，论文中也提出了新的树的组织结构，能够进一步提高树中节点的扇出。在测试方面，我们使用 gem5 进行模拟，测试结果显示在大多数场景下，可挂载的完整性保护树能够减少 2 10 倍的性能开销，为云场景下的内存完整性保护提供了可能。

总体来说，本文的贡献如下：

- 分析了已有完整性保护方案的优缺点，以及在云场景下的不足与缺陷
- 提出了可挂载的完整性保护树，能够支持 TB 级别的内存以及内存的动态保护
- 在 gem5 模拟器上进行模拟，得到了 2-10 倍的性能提升。

1.3.1 论文组织结构

本论文得组织结构如下：

第一章介绍现阶段相关的研究工作，对各种不同得完整性保护方案做了系统得介绍以及对比。

第二章　相关技术背景

2.1　Merkle Tree 及其密码学基础

Merkle hash tree[1] 算法首先是由 Ralph C. Merkle 于 1987 年在 “A DIGITAL SIGNATURE BASED ON A CONVENTIONAL E;UCRYITION FUNCTION” 提出，在该论文中首次提出了仅仅依赖于传统加密方法的数字签名系统，而实现该系统的核心就是 Merkle hash tree 和 Lamport 算法。Merkle hash tree 算法能够快速查询并且检查签名值是否正确，并且仅仅只需要使用少量的内存（与总内存呈对数相关）。签名的大小往往在几百个 byte 到几千 byte 之间，而生成签名则需要经过多次的底层哈希计算。通过采用了 Merkle hash tree 算法起先广泛的使用在签名系统中，以保证签名数据的完整性，之后逐渐在网络和文件系统中得到了广泛的应用。近年来，随着计算机可信执行环境的演进和非易事性内存的商用化，以 merkle hash tree 为原型的完整性保护算法在在保护运行时内存完整性上起到了关键的作用。

2.1.1　单向散列函数与一次性签名

2.1.1.1　单向散列函数

单向函数是正向计算非常简单，但是逆向计算非常困难的函数，例如给定一个函数 F，给定一个输入 x，计算 $y = F(x)$，非常容易，但是确定 x 的值使得 $F(X) = y$ 非常困难。单词散列函数是基于传统的密码学加密函数观察所得：即给出明文和密文，想要推导出对应的私钥是非常困难的。如果我们定义一个传统的加密函数 $S($[illegible]$) =$ [illegible]，那么我们可以类比定义一个单向函数 $F(x) = y$，它等价与 $S(x, 0) = y$, 其中我们对一个常量用秘钥 x 进行加密，得到了密文就是单向函数的输出。如果给出 y 推导出 x 那么就等价于我们知道明文是 0 密文是 y 就能够推导出秘钥是 x，显然这和传统的加密函数的观察结果不符。

单向散列函数，是单向函数中的一种，它能够接受任意长的输入（几千比特），然后产生定长的输出结果（64 比特）。在使用单向散列函数时候需要非常小心，因为可能单向散列函数存在“平方根”攻击等安全漏洞，对于一个 54 bit 的数据，攻击可以通过 2 的 28 次操作之后能够生成相同的结果。当然在大多数的场景下，此类攻击的代价会超过攻击对象本身的价值，因此一个良好设计的单向散列函数一般认为是安全的。单向散列函数具有一下几个优点：

- 对于任意长度的输入，可以生成固定长度的输出结果
- 相较与求幂取模签名计算来说，计算更加快速，并且能够很方便的整合到硬件的设计之中
- 输入数据不同，得到的散列值也不同
- 具有单向传递性

2.1.1.2　一次性签名

一次性签名最早是由 Lamport[2] 在 1979 年所提出，Lamport 经过观察发现，单向散列函数能够提供非常强大的签名方案与系统。下面我们将从签名单比特数据和签名多比特数据分析：

单比特签名：对于 A 签名一个单比特数据给 B，首先 A 先随机生成两个值 x[1], x[2]，然后我们选择一个单向散列函数 F，对 x[1], x[2] 进行计算得到对应的 y[1], y[2]。然后我们将 y[1]，y[2] 作为公钥暴露给 B，而 x[1], x[2] 最为私钥保留在 A 手中。对于一个单比特数据来说，如果是“0”，那么我们选择 x[1] 作为签名个 B, 如果是“1”选择 x[2] 作为签名给 B。假设现在单比特消息为“1”，那么 B 能收到的签名 x[2]，B 可以通过单向散列 F 验证 F(x[2])，是否等于 y[2]，因为在这里 F 和 y[2] 都是公开的，所有人都能够去验证这个结果。但是只有 A 同时知道 x[1], x[2]，根据单向散列的函数的原理，知道 y[1], y[2] 的 B 无法推导出 x[1],x[2]。然而一次签名并不是完美的，它只能够签名一次，因为一旦签名一次之后，B 就能够知道 x[1], x[2] 中的一个值。B 可以通过让 A 签名不同比特的消息获取 A 中所有的私钥，这也是之后提出 merkle tree 的一个主要出发点。

多比特签名：多比特签名是在但比特签名的基础上演进而来的，对于长度为 m 的消息来说，分别随机生成 x[0:m], x[m, 2m]; 同时选定一个单向散列函数 F，分别对 x[0:m], x[m, 2m] 做计算出 y[0:m], y[m:2m]。跟单比特相似，x 作为 A 的私钥而 Y 作为公钥发布给验证签名的 B。对于 M 中的每一个比特，都经过类似单比特的计算，如果是“0”就从 x[0:m] 中选择，如果是“1”就从 x[m:2m] 中选择。之后可以得到签名后的数列 s[0:m]。将 s 和 M 一起传给需要验证的 B，B 根据 s 中每个比特的数值，用公钥 y[0:m], y[m:2m] 对 s[0:m] 做验证。因为单向散列函数 F 的特性，保证了验证者不会通过 y 推导出私钥 x。当然多比特签名仍然存在只能签名一次的问题，并且对于任意消息 M，都需要返回一个等长的签名 s 给签名的验证着 B，并且对于 A 来说需要生成两倍与验证消息的公私钥对 x, y。一种优化的方式是，只生成等长与验证消息的 x, y。其中如果消息中的比特是“1”，我们就用 x 签名，如果是“0”就用 0 签名。对于这样的优化，我们可以减少一般的秘钥空间开销。但是显然这样的设计是有问题，例如对于消息“01001110100”，验证者会收到 x[2], x[5], x[6], x[7] 和 x[9]。但是 B 不能够保证它收到的消息一定是完整的，恶意的中间攻击者可以将第二个比特从 1 改成 0，同时传给 B x[5], x[6], x[7] 和 x[9]，对于 B 来说他并不能意识到该消息已经被人篡改，签名是无效的。为了解决这个问题，需要在签名中加上该消息中“1”的个数或者其他哈希值，保证消息的完整性，并且将该值和原消息一起被 A 签名，然后发布给验证者 B。虽然该优化能够减少一半的空间开销，但是对于一个想验证大量消息的 B 来说，他还需要存储和验证消息相同数量级的公钥 y，这对于验证者来说是非常不友好的。

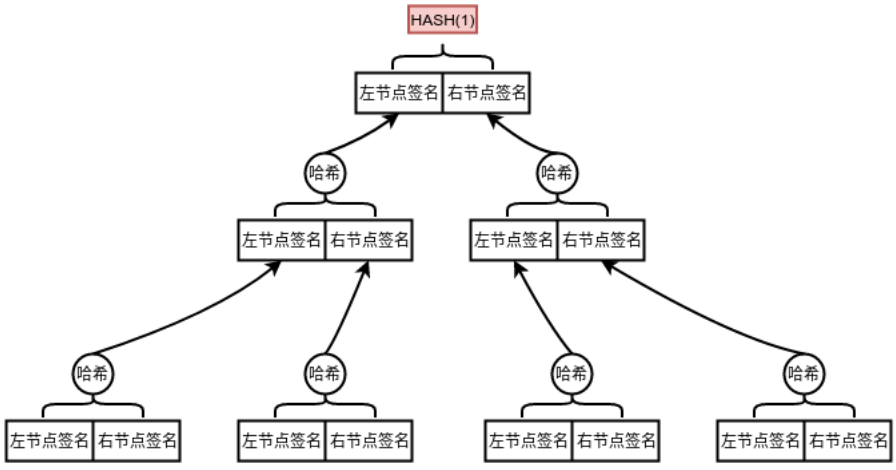

图 2–1 **Merkle Tree** 每个节点保存着左右子树 hash 的签名

2.1.2 Merkle Tree

为了解决上面所说的验证着需要存储大量的公钥的问题，Ralph C. Merkle 提出了一种新颖的数据结构 merkle tree，它能够大大的减小验证着需要保存的公钥的大小，从而减少验证者的开销。再论文中，merkle tree 是用来进行前面验证，随着 merkle tree 的不断发展，也被广泛的应用再完整性保护中。

如图 2–1所示，我们以二叉树为例，在签名系统中 Merkle tree 中的每个节点保存着三个签名，分别是：左节点认证，右节点认证以及数据的签名。根据二叉树的组织形式，我们很容易根据父节点推到出子节点的标号，例如父节点的编号 i，那么他的字节的编号分别是 2i，2i+1；同理，如果我们知道子节点的编号我们很容易知道它的父节点的编号，这一点不论在签名系统还是完整性保护系统中都显得非常的重要。因为验证者可以提前知道每一个节点以及其父子节点的编号，而不需要通过获取父节点之后才能够计算出其子节点的编号。对于一个节点中的，左右子节点认证以及消息签名，分别由三组私钥 x，公钥 y 完成签名认证。同时我们选择一个合适散列函数，对该节点中的公钥进行哈希计算，得到的哈希值由其对应的父节点进行签名认证。通过这样的机制，可以大大减少验证者需要知道的信息的大小。在 merkle tree 的验证中，验证者们只需要知道根节点的哈希值，当验证者需要验证第 i 条消息是否被签名者签名过，签名者需要讲第 i 个消息，以及对应的公钥发给验证者。然后签名者通过自己的私钥，对需要签名的消息进行签名，并且把签名后的结果发给验证者。验证者通过公钥验证签名结果是否正确，如果通过验证，验证者将该节点所有的公钥的哈希值以及对应的父节点公钥发给用户，验证者可以通过父节点中的公钥验证哈希值是否正确。如果当前的节点已经是根节点，因为根节点的哈希值被所有的验证者保存，验证者通过将计算得到的根节点哈希值和自己保存的根节点哈希值做对比，如果两者一致，那么整个签名的过程得到验证；反之，签名验证失败。通过 merkle tree 的组织形式，验证者所需要知道的共识仅限于根节点的哈希值，另外在验证过程中，签名者会向验证者发送仅仅对数于总消息数量的信息，用于辅助验证签名的合法性。

merkle tree 最初使用在签名的系统中，但是之后被广泛的应用在了完整性保护中，和签名系统中的结构相似，一、需要保护数据的完整性；二、需要保证验证者只需存储少量的信息。在完整性保护架构中，保护的数据被分成不同的数据块，采用 merkle tree 保护所有数据块的完整性。在数的叶子节点中，存储着所有的数据块的哈希值，而父节点中存储着所有子节点的哈希和得哈希。数据块得完整性由叶子节点的哈希值保证，所有叶子节点哈希值得完整性由父节点保证，一次类推，我们只要保证根节点得哈希不被篡改，即可以保证数据块中数据得完整性。以 merkle tree 得形式保证数据得完整性，由一下几点好处：

- 需要存储得可信数据很小：根节点哈希，其余节点可以存在不可信的存储中
- 可以并发进行数据哈希校验，每个数据块的哈希不依赖其他数据块，可以同时进行
- 支持数据更新后，只需要更新对应部分的哈希值即可，不需要对所有的数据进行重计算
- 对单个数据块完整性验证的次数和数据块的总数成对数相光

除了 merkle tree 的结构，还存在链表等结构，但是相较于 merkle tree 而言，无法并发进行完整性保护验证，以及不支持动态更新，在这里就不做更多的阐述。

2.2 Bonsai Merkle Tree

Bonsai Merkle Tree[3] 在 Merkle Tree 上做了进一步的优化，Bonsai Merkle Tree 保留了 Merkle Tree 中的树状组织结构，继承了 Merkle Tree 中只需要保存少量的根节点哈希，能够针对少量的更新快速的重新计算哈希等等优势。但是 Merkle Tree 还是存在一些较为严重的问题，例如树节点的扇出较少，无法和内存加密保护结合。因此在 AISE 中提出了新的完整新保护的数据结构 Bonsai Merkle Tree，不同于 hash tree，Bonsai Merkle Tree 是以计数为基础的 counter tree。hash tree 节点中保存的是子节点的 hash 值，hash 虽然能够保证数据的完整性，但是 hash 的大小会有严格的限制，因为 hash 大小决定了哈希碰撞的概率，从而限制了一个树节的扇出。树节点的扇出较少在少量的信息保护的场景或者不关心检查开销的程序来说适用，但是在内存完整性保护上，程序非常关心实时的检查开销，树节点的扇出较少会导致树的层数较深，实时完整性检查的开销也会随之增加。一个简单的想法是增加树的扇出，减少树节点中哈希的大小，因此在 BMT 中提出了基于计数的 counter tree。

2.2.1 内存计数

内存计数：counter 是内存加密与完整性保护中的一个重要的概念，counter 增加了内存时空维区分，来防止重放映等攻击。counter 首先被使用在内存加密中，用于生成和时空相关的加密 pad，之后又于完整性保护整合，作为保护对象的版本。

基于计数的内存加密内存加密是内存保护中重要部分，传统的内存加密使用复杂的内存算法，会带了较为严重的运行时开销，因此在 AISE 中提出了一种基于 counter 的一次性内存加密。基于 counter 加密能够充分利用给时空因子，从而使一次性加密成为了可能。如图 2–2所示，在 AISE 中，作者使用了 AES 算法作为加密引擎，AES 算法是一种高效的加密算法，只需要经过十轮矩阵的异或运算就能够得到相应的密文。然后单纯只是用加密算法是不够的，因为如果使用相同的 key 和相同的明文，得到的密文是一致的。在内存加密中，攻击者可以同时知道密文与明文，因为密文与明文之际俺的关系不会发生变化，所以攻击者可以建立起明文与密文你的对应表，从而获取内存中的数据。因此在 AISE 中作者没有简单的将明文用 AES 进行加密，而是使用 AES 生成加密的 pad。用 pad 与明文做异或的操作，从而得到密文；而解密的过程也是一致的，因为异或操作具有交换律，所以我们对密文做异或，就能够得到相应的密文，特别的异或操作往往只需要一个 cycle 就能够完成，并不会增加加密的开销（相较于 AES 加密，虽然 AES 已经是一个高效的加密算法了）。使用 pad 对明文的数据进行加密是一个高效的加密方式，但是只能进行一次加密，因为如果攻击者同时知道明文和密文，就能够推断出加密用的 pad，从何用 pad 对其他的数据进行加解幂。所以我们需要保证 pad 尽量只能使用一次。为了解决这个问题，我们为 pad 的生产加入相应的时空因素，其中我们使用 counter 来表示 pad 的时间因素，用访问内存地址来表示 pad 的空间因素，对于相同地址的内存的每一次访问，都会是 counter 加一，从而保证了不可能出现两个时空因子都相同的 pad。pad 是由 AES 加密引擎生成的，通过 AESk 密钥，对由 counter 和 addr 拼接而成的 128 位 seed 进行加密，生成加密的 pad，通过 pad 对明文进行异或得到对应的密文。

计数（counter）在内存加密中保证了加密 pad 具有一定的时空因素，也正是因为具有时空因素，所以可以使用一次性加密。在完整性保护中，完整性保护引擎巧妙地复用了 counter，将 counter 作为完整性保护中改的版本信息，从而防止重映射等攻击。在 Merkle Tree 中我们主要介绍了对消息的完整性保护机制，但是没有结合内存的相关知识，这里我们将以 BMT

为例，对如何保护内存的完整性，做进一步的阐述。

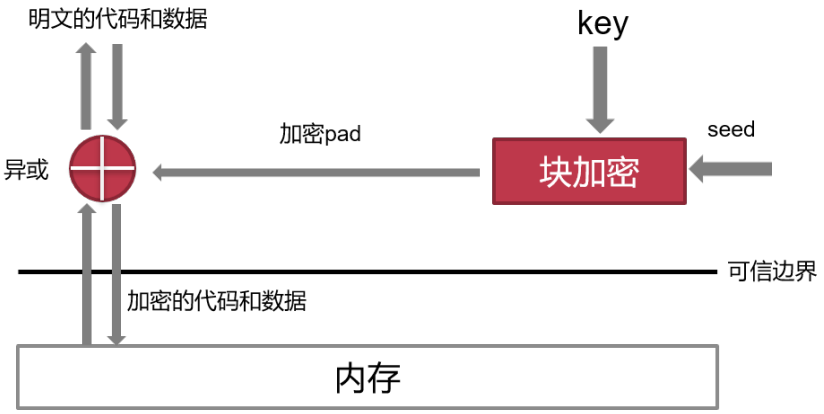

图 2–2 基于计数的内存加密

2.2.2 针对内存的完整性保护

Merkle Tree 提供了一种巧妙地数据结构，通过树地形式保护消息地完整性。正如在之前章节所说地，Mekrle Tree 的一大优势就是需要保护的可信数据非常少，只需要保护根节点中数据就可以了。在内存完整性保护中，内存被认为是不可信的，只有保存在 SoC 中的数据才是安全的（攻击的难度较大）。因为 SoC 中的空间大小有限，能够保存的可信数据远少于内存中的数据。Merkle Tree 的巧妙的数据结构恰好能够解决 SoC 中空间有限的问题，因为只需要保护根节点中的数据。

内存攻击模型针对内存完整性的攻击主要分为三类：欺骗（spoofing），拼接（splicing）以及重放映（replay）。内存欺骗是指攻击者可以伪造内存中的数据，替换内存中现有的数据；内存拼接是指将其他地方的内存数据移动到受攻击的内存位置中；内存重放映是指将相同的内存地址上的老版本的数据重新写入该内存中。其中前两类攻击可以通过计算 hash（HMAC）来防御。因为攻击者无法获得 hash 算法的 key，即无法通过已知的输入（counter，addr，明文）生成对应的合法的密文。又因为内存欺骗，和内存拼接会改变内存数据或者内存地址，导致输入三元组（明文，counter，addr）至少有一个发生了变化，所以新的输入三元组是从未出现过的，自然对应的密文也无法被攻击者直接获取。使用 hash 算法（HMAC）可以很好的保证内存的欺骗，以及内存的拼接攻击，但是无法抵御内存重放的攻击。内存重放是指将同一个内存地址上的旧的数据重新加载到该内存中，在内存重放的攻击中输入三元组（counter，addr，明文）在过去曾经出现过，所以对应的密文攻击者也可以获取。攻击着可以将内存中的数据替换成为老版本的数据，来实现 rollback 的攻击。为了防御这类攻击，我们将 Merkle Tree 中的根节点放在 SoC 中，在 SoC 中的根节点处在攻击范围之外是可信基础，所以一旦采用重放攻击，需要替换对应物理内存中所有的 counter，其中包括了根节点的 counter，因此攻击者无法完整重放攻击。自此，结合 Merkle Tree 的内存完整性保护能够很好的保护物理内存免受拼接，欺骗以及重放的攻击，结合 SoC 上保护的根数据，能够实现对内存的完整性保护。

2.2.3 BMT 数据结构

正如之前所说的 Merkle Tree 存在节点的扇出较少，树的深度较深从而导致无论是运行是的开销还是内存的开销都较大（运行时开销是因为树的深度较深，检查的次数较多；内存开销较大是因为树的扇出较小，单个 hash 保护的数据大小是固定的，当数据规模较大时候，存储 hash 的开销也会随之增加）。在 BMT 中结合内存加密，设计了新的数据结构，希望缓解不论是运行时候的开销还是内存开销。研究人员发现：

1. Merkle Tree 的设计是为了防止重放的攻击，其他的类似于内存拼接与内存欺骗可以通过计算 hash 对到对应数据块的 MAC 来防御
2. 已有的内存加密都使用了基于 counter 方式，因为 counter 会在写内存的时候被怎家，所以也可以被当做数据块的版本号

针对上面两个观察，研究人进一步提出，如果能够做到一下三点，那么数据不需要被 Merkle Tree 保护：

- 所有的数据块都有自己的 MAC，通过使用 hash 函数计算而得
- 计算 MAC 需要使用到内存块的 counter 和 addr
- counter 的完整性得到保护

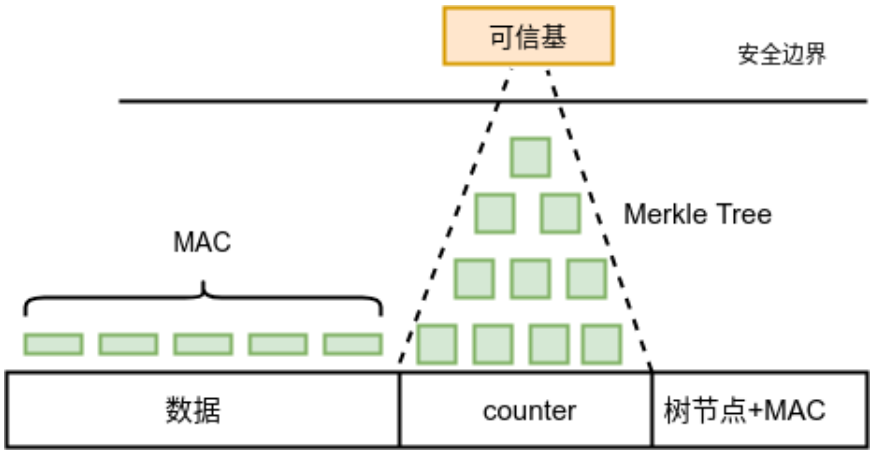

图 2–3 **Bonsai Merkle Tree**：对 counter 的完整性保护

如图 2–3所示，根据上述三个条件，我们发现可以通过对 counter 的完整性保护来代替对数据的完整性，因为 counter 的大小为 7 bit，远小于 hash 的 64 bit，所以对应的 Merkle Tree 所需要的额外内存就更少，同时因为一个 cache line 中可以存放更多的 counter，使得树叶子节点的扇出更大，减少了树的层数，提升了运行时的性能。虽然保护 counter 能够比保护数据来的更有优势，但是我们还需要论证仅仅保护 counter 是否能够起到和保护数据一样的安全性，下面我们将从数学的角度做严格的证明

counter 完整性保护的证明：我们用 P 和 C 来表示数据块的明文与密文，数据块对应的 counter 用 ctr 表示，对应的 MAC 用 M 表示，hash 函数 H 中使用的秘钥为 K。数据块的 MAC，通过使用秘钥的密码学哈希函数 H 对输入密文 C 和 counter 计算得到，例如：$M = H_K(C, ctr)$。完整性验证时候会计算 MAC 并且和之前已经计算的存储在内存中的 MAC 进行比较，如果他们之间不匹配，那么完整性验证将失败。因为 counter 的完整性被保证（我们之前声明过），攻击者不能够改变 ctr 却不被发现。攻击者只能够将密文 C 篡改成 C'，或者将 MAC 改成 M'。但是因为攻击者不知道哈希函数 H 的 key，所以攻击着无法生成与 C'

匹配的 M' 同时因为密码学哈希函数的不可逆性，也无法根据 M' 生成与之匹配的 C'，所以 $M' \neq H_K(C', ctr)$，从而无法通过内存的完整性检查。另外攻击者也无法将 C' 和 M' 同时替换成之前的老版本，因为 $M^{old} = H_K(C^{old}, ctr^{old})$，但是 ctr 受到完整性保护，能够保证它一定是 ctr 和不是 ctr^{old}。因此重放攻击也会被防御。

经过上面的证明，可以得出只需要保护 counter 的完整性同时就能够保证内存数据的完整性，又因为一个 512 bit 的数据对应的 counter 只有 7 bit，因此需要完整性保护的数据将显著的减少，对应的 Merkle Tree 的大小和层数也大幅度减少，从而提升了运行是检查的性能缓解了内存的开销（Merkle Tree 从需要覆盖内存数据和 counter，减少为了只需要覆盖 counter。

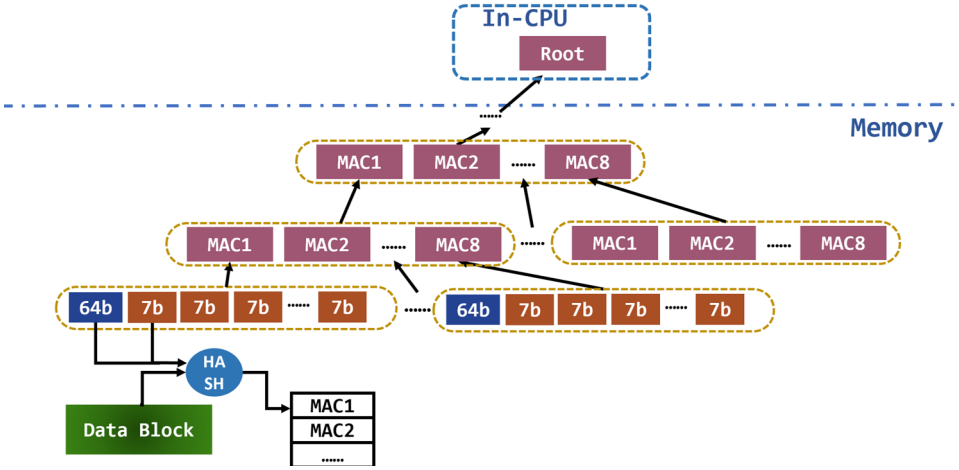

图 2–4 **Bonsai Merkle Tree** 叶子节点中保存了 64 个 7 bit 的 counter

如图 2–4所示，BMT 叶子节点中存储了 64 个 7 bit 的 counter，每一个 counter 对应一个数据块。通过密码学 hash 函数，将 counter 和数据块内容进行计算生成 MAC，与存储在内存中的 MAC 进行匹配。为了保护 counter 的完整性，BMT 在 counter 上建立 Merkle Tree 来保证所有的 counter 无法被攻击者修改。每一个 hash 为 64bit，一个树节点中能够存放 8 个 hash，因此 Merkle Tree 的扇出为 8。

2.3 VAULT

在 BMT 中，研究者发现结合基于 counter 的内存加密，完整性保护时只需要保护内存的 counter 而不需要保护内存数据本身，从而减少了 Merkle Tree 的内存及运行时的开销。但是 BMT 对完整性的保护还是基于 Merkle Tree 的形式。所以并没有改变 Merkle Tree 本身的数据结构，对于 counter 的保护可以理解成只增加叶子节点的扇出（从 Merkle Tree 扇出 8 增加到了 64）。然后只增加叶子节点的扇出显然是不够的，当需要保护的内存到达 GB 级别，叶子节点的扇出并不能起到明显的优化效果，所以我们需要尽可能的增加树中每一层的扇出。因此研究者在 BMT 的基础上做了进一步的拓展与优化，从而实现了对 GB 级别的内存完整性保护。

2.3.1 全局计数与本地计数

在 BMT 中采用了对 counter 的完整性保护，被认为是一直高效的完整性保护方式，但是不幸的是，在 BMT 没有在树的每一层节点中都使用 counter，而只在叶子节点中将 hash 替换成了 counter。因此在 VAULT 中做了一次大胆的尝试，能不能将树中每一层节点中的 hash 都替换成 counter，这样能够提高非叶子节点的扇出。然后考虑到完整性保护树的结构特性，越在顶部的树节点越容易被访问到，所以 counter 增加的也越快，出现 overflow 的频率也越高。我们回顾一下在 BMT 中使用的 counter 的大小位 7bit，因此对统一内存写 2^7 之后就会将 counter 全部耗尽。counter 的耗尽意味着攻击者有概率发起重放攻击，攻击者可以将内存改写成为 2^7 个版本之前的数据（因为 2^7 个版本前的内数据和当下的数据拥有相同的版本号），更少的 counter bit 意味着带来更大的安全风险，所以该如何兼顾安全与性能呢？在 VAULT 给出了一种可行的解决办法。在 VAULT 的树节点中，存储着两类的 counter：（a）全局的 counter；（b）本地的 counter，所以一个完整的 counter 是由一个共享的全局 counter 加自己都独有的本地 counter。在一个树节点中（512 bit），存在一个共享的全局 counter（64 bit），一个树节点 hash 值（64 bit）以及 N 个本地的 counter（$N * nbit$）。这样设计的好处是全局的 counter 解决了 counter overflow 的问题，因为全局的 counter 足够大（64 bit），如果要将全局的 counter 耗尽需要做 2^64 次写操作，这个在一台机器的生命周期中基本不可能实现；而本地 counter 决定着树节点的扇出，相较于全局的 counter，本地 counter 只拥有少量的 bit 数（在叶子节点中只有 6 bit，可以存放 64 个本地 counter），本地 counter 的数量决定了树节点的扇出，高扇出能够降低树的深度，减少运行时完整性检查的开销。另外考虑到树本身数据结构的特点，越往上的树节点越容易被访问到，在 VAULT 中不同层的树节点中本地 counter 的大小和个数并不相同。例如在最底层（叶子节点）中本地 counter 的大小位 6 bit 个数位 64 个，而在倒数第二层中本地 counter 的大小 12 bit 个数位 32。本地 counter 的大小不需要一直增大，因为越往上的 counter 越容易被 CPU 的 cache 缓存住，被 cache 缓存的 counter 被认为是安全的，当发生一次写操作的时候可以不需要增加 counter 的值，从而缓解较高的 counter 增加过快的问题。在 VAULT 中本地 counter 最大为 24 bit，个数为 16，因此在 VAULT 中树节点最小的扇出是 16，相较于 Merkle Tree 的扇出 8 来说有了一倍的提升。当保护的内存较小的时候，底层树节点的扇出更大，对性能的提升也更加的明显。

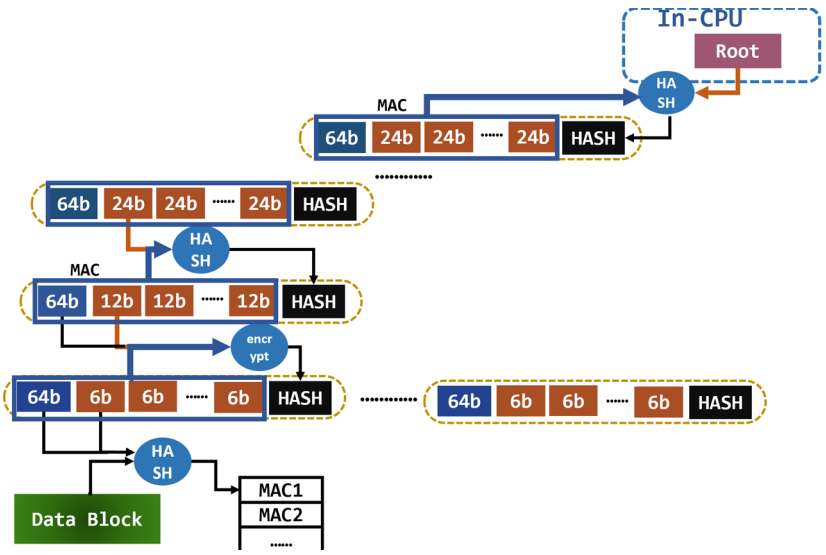

图 2–5 **VAULT** 用 counter 替换 hash，树节点的扇出分别为 64, 32, 16

2.3.2 基于 counter 的完整性检查

如图 2–5所示，在 VAULT 中每一层树节点的扇出分别是：64，32，16，...，因为现在完整性保护树存储的 counter 和不是 hash，所以应该如何保护 counter 的完整性呢？在上面我们提到一个树节点中除了全局的 counter 和本地的 counter 之外，还存由一个 64 bit 的 hash，那么如何使用这个 hash 来保护 counter 的完整性呢？根据 Merkle Tree 的思路，所有子树的完整性由其父节点保护。在基于 counter 的完整性树中，将父节点中对应的 counter 和该节点中所有的本地 counter 和全局 counter 分别作为 hash 函数的输入，计算得到相应的 hash 值，存储在该节点中（即 64 bit 的 hash）。这个 hash 的生成包括了该节点所有 counter 的信息，以及父节点中对应 counter 的信息，只要保证父节点钟大哥 counter 的完整性，那么攻击者无法通过修改该节点中的 counter 值来生成合法的 hash 值，下面我们将具体介绍读操作和写操作的完整性检查与更新。

读和验证：当发生一次读请求时，内存控制器首先会读取内存数据，内存数据对应的 MAC 以及完整性保护树中的叶子节点。内存控制器会根据叶子节点中的 counter 以及内存数据进行哈希计算（HMAC）将计算得到的 MAC 值与存储的 MAC 做匹配，如果不匹配，那么完整性保护检查失败。之后内存控制器会去索引叶子节点对应的父节点中的 counter，将父节点中的 counter 和叶子节点中所有的 counter 进行哈希计算，得到对应的 MAC 值，将计算的到的 MAC 值与存储在叶子节点中的 MAC 值进行匹配，如果不一致，那么完整性保护检查失败。重复上诉步骤，直到父节点的 counter 处在 SoC 中，或者父节点即为根节点，则完整性保护检查结束，验证通过。

写和更新：当放生一次写请求时，内存控制器奖新数据写到对应的内存块中，然后增加完整性保护树中的叶子节点 counter 数值。之后内存控制器中的完整性保护引擎根据新的内存数据和增加的 counter 计算出新的 MAC 值，然后将更新后的 MAC 值写回到内存数据对应的 MAC 中。之后内存控制器会增加叶子节点对应的父节点中的 counter 值，同时将叶子节点中所有的 counter 和增加过的父节点 counter 进行 hash 计算得到新的 MAC 值，将新的 MAC 值写入叶子节点中存放 MAC 的区域。重复上诉步骤，直到当前节点的 counter 已经在 SoC 中，或者父节点为根节点，则一次写操作的更新完成，同时为写入的新数据提供了完整性保护。

第三章　可挂载完整性保护树：Mountable Merkle Tree

siunitx ntheorem amsthm biblatex-gb7714-2015

在前面几张中，我们详细介绍了已有的完整性保护方案，包括了最新颖的 VAULT[4]。这些工作的一个共同点是希望通过增加树中节点的扇出来减少树的深度，同时进一步优化运行时候的检查的开销。然后盲目增加树节点的扇出回带来安全隐患（replay 攻击等等），在最新颖的完整性保护方案 VAULT 中，树中节点的扇出 24（第三层及以上），同样无法保护 TB 级别的内存数据。在本文中，我们提出了一种新的完整性保护数据结构：哈希森林以及可挂载完整性保护树（Mountable merkle tree）。希望通过新的完整性保护数据结构，能够实现一下几点：

- 支持 TB 级别的内存完整保护
- 支持动态的内存完整保护分配机制
- 支持离散的内存保护
- 可以接受的运行时开销
- 实现保护内存的可扩展性

可挂载完整性保护树（MMT）以及哈希森林，能够很好的解决当下安全领域一些热点问题：enclave 的内存完整性保护以及 NVM 中的数据保护。同时该方案也能够和已有的完整性保护结合，具有较好兼容性。

3.1　哈希森林, 子树与根树

已有的完整性保护方案都采用了哈希树的方式管理哈希或者计数，树的组织结构能以较小的开销，保护大量的内存数据；但是随着需要保护的数据越来越大，哈希树只能够通过增加树的深度来保护更多的内存，然后增加树的深度会带来更大运行时的性能开销，这也是之前工作的共同缺陷所在。我们认为，增加保护的内存大小不仅仅只能通过增加树的深度来实现，如果增加树的个数，同样也能够保护更多的内存，同时也不会增加运行时的开销。在本篇论文中，我们提出了哈希森林的概念，哈希森厉中由一组哈希树构成，每一颗哈希树都能够保护一块物理内存，如果需要保护更多的物理内存，只需要网哈希森林中增加更多的哈希树即可，同样如果需要保护的内存减少，那么只需要将哈希树从哈希森林中移除。哈希森林解决的两个问题：一、时如何动态的增加活减少保护的内存的；二、通过增加树的个数来增加保护内存的大小；三：支持可扩展的完整性保护。哈希森林的一种简单的实现，是保护更多的哈希树的根节点，然后正如我们之前所提的，验证者不希望保护大量的数据，对比于完整性保护中，即内存控制器（memory controller）中的空间有限，只能保存少量的哈希值活着计数值所以如果仅是简单保护更多的哈希树根节点，显然没有办法达到期望的具有可扩展性的内存保护，因为保护的内存大小收到内存控制器中能够保存的根节点数量的限制。为了实现可扩展的内存保护，我们提出了两个新的概念：子树和根数。我们认为整个哈希森林由子树和根树组成。子树即哈希森林中所有的可以动态加入或者移除的哈希树，一颗颗子树构成了哈希森林；根数则保护哈希森林的完整性。所以我们只需要把根数的根节点：root-of-root 保护在内存控制器中即可，而不需要保护所有的子树根节点。因为内存控制器中的空间有限，我们无法将所有的子树节点放在内存控制器中，相反我们把所有的子

树的节点以及根节点保存在内存中，我们将所有子树的根节点放在一块特殊的内存空间中：MMT 元数据区域（之后会详细介绍）内存中的空间被认为是充足的，我们可以存储足够多的子树。当然在内存控制器中仅仅只放根数的根节点是不够，这样会退化成为之前的完整性保护方案，即仍然只存在一棵树。为了讲哈希森林结合到内存控制器中，我们首先对内存的访问做了调研。我们发现，CPU 对内存的访问具有一定的局域性，即在一段时间内，CPU 可能访问某一部分的内存数据，而不会随机的访问整个内存空间。这个观察结果已经广泛的使用在了计算机体系结构中，比如 cache 的提出就是为了缓解 CPU 频繁的访问内存中的数据，既然内存中的数据可以通过 cache 的提升性能，那么我们也可以采用同样的方式缓存子树的根节点。我们首先将所有的子树的分为三类：一、活跃的子树，即 CPU 正在访问该子树保护的内存数据或者刚刚访问了该子树保护的数据；二、不活跃的子树，该子树对应的内存数据需要被保护，但是当前或者近段时间内，该内存数据没有被 CPU 访问；三、是未分配的子树，未分配的子树表示内存的数据不需要经过保护。首先活跃的与不活跃的子树的节点都需要在内存中有备份，如果该子树是活跃的，那么我们将该子树的根节点缓存在内存控制器中，CPU 可以直接在内存控制器中读到子树根节点的哈希值或者计数值。如果 CPU 访问的子树当前活跃，我们在后续段落中做京一部的讨论。综上，我们提出了哈希森林，子树和根数等的概念，我们通过增加哈希森林中子树的个数来保护更多的内存。为了节省内存控制器中的空间，我们只需要保护根数的根节点，为了在大多数检查中只需要检查子树的节点，我们将当前活跃的子树根节点缓存在内存控制器中，这样对内存的检查不需要从根数的根节点开始，只需要从子树开始即可，来实现增加内存大小，但在大多数情况下不增加检查时访问树的深度。下面我们将从内存完整性保护架构中的不同部分，以及 MMT 的管理机制等方面做具体的阐述

3.2 内存控制器

内存控制器是控制对内存的访问，内存控制器是 SoC 上的一个单元，CPU 中的访存指令在 cache 中如果没有命中，或者 cache 中发生了 cache miss 等事件，需要从内存中读取或者写入数据的时候，就会向内存控制器中发送一个包的请求。内存控制器解析包请求，然后通过总线将需要读取或者写入的数据的物理地址（数据）以包的形式发给 DIMM，DIMM 中的控制器相应请求，写入或者返回对应物理地址上的数据。在一个 CPU 上可以拥有多个内存控制器单元，每一个内存控制器通过总线控制一个 DIMM，通过多个内存控制器，能够实现并发的内存操作，CPU 会协调不同的内存控制器，使其能够高效协同工作。为了在访问内存的时候保证内存数据的完整性，我们需要在内存控制器中做一定的修改。传统的完整性保护架构中，只需要在硬件中实现对完整性保护树的索引，取值，更新与哈希计算，在 MMT 中因为引入了哈希森林，所以还要加上对子树的管理，分配等其他功能。在 MMT 的内存控制器中我们增加了三个组件：1、安全比特表 (Secure Bitmap)；2、子树缓存（Mount table）；3、根树根节点哈希/计数（Root-of-root），如图 3–1所示。

安全比特表：Secure Bitmap 安全比特表用于记录那些子树已经分配了，在之后的章节中我们会详细的介绍内存中的布局。安全比特表用一个比特来代表其对应的子树是否被分配，活跃的子树以及不活跃的子树都是已经分配的子树，而未分配的子树是不活跃的子树。安全比特表的开销取决总内存的大小以及子树的大小，考虑到一颗能够保护 4M 大小的子树以及总内存为 512G，，那么对应内存控制器中的安全比特表的大小只需要 16K。16K 的大小对于现在内存控制器来说是完全可以接受的，如果子树的保护的内存更加的广泛，那

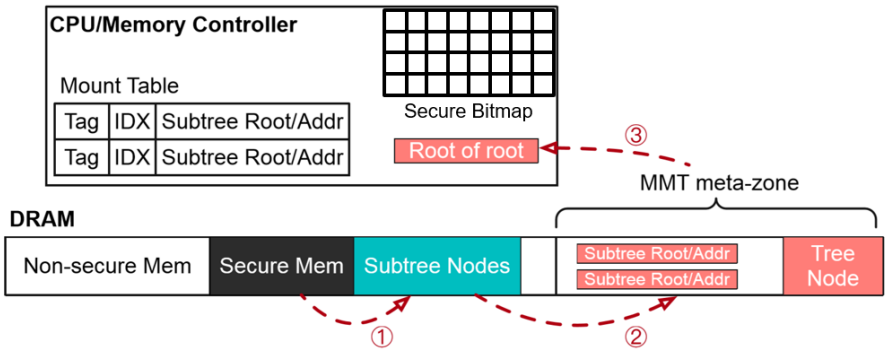

图 3–1 **MMT** 中内存控制器与内存布局。*Secure Bitmap* 记录了当前已经分配的子树，*Mount Table* 记录当前活跃的子树根节点，*Root-of-root* 根树的根节点 (可信基)。内存布局：*Non-secure Mem* 非安全内存，*Secure Mem* 安全内存，*Subtree Node* 子树节点，*MMT metadata zone* MMT 元数据区域

么安全比特表的开销将进一步的减少。当安全比特表中的比特为 1，说明对应的子树已经分配，需要完整性保护的检查，如果未被分配，说明是正常的内存，不需要进行完整性的保护。通过安全比特表这个概念，我们可以实现动态的内存安全内存的分配，而不需要事先指定需要保护的内存区间。安全比特表也可以被其他的组件所代替，比如 CPU 对内存的访问可以带上一个特殊的标志位，如果该标志位是 1 则说明访问的是需要完整性保护的内存，反之则是正常的内存。通过这样的机制我们可以实现更加细粒度的内存区分，在之后的章节中我们会详细的介绍如何按照页的粒度进行完整性保护。

子树缓存：Mount Table 参考内存中缓存的数据结构，我们在内存控制器中增加子树缓存。子树缓存类似与内存中 cache 的结构，也分成多个集合，每一个集合中又包括了多路的子树缓存行，每一个子树缓存行中存在对应的 tag，index，以及 4 个条目的子树根节点。同时每一个子树缓存集合中还包括了 LRU 等相关信息，用于子树的驱逐策略。子树的根节点主要有两部分组成的，一部分是子树哈希/计数值，另一部分是子树在内存中所处的位置。之所以需要增加一个子树在内存中所处的位置，是因为子树是可以动态的分配的，我们不能事先给子树确定一块内存空间，所以当一条内存访问指令来的时候，我们需要通过子树的地址来确定子树在内存中的位置，从而通过子树进行完整性保护的检查。子树的根节点大小总共为 16byte。首先子树根节点的大小必须能够整除一条缓存行的大小。在这里缓存行的大小为 64B，同属内存控制器也是按照 64B 的大小进行内存的读写操作，所以 16B 刚好是一条缓存行的四分之一；另外按照实际上更节点中的计数值以及哈希/计数值的大小可以压缩到 8 byte，但是考虑到我们方案的通用性，以及之后的延展性，我们这里将一颗子树的根节点大小设置为 16 byte。当 CPU 向内存控制器发送一个请求的数据包的时候，内存控制器能够解析该请求包。首先内存控制 2 器会将该请求的物理地址转换为对应的内存块。正如之前所说的，缓存行的大小以及内存块的大小都是 64B，所以我们对内存的读写都需要以 64B 对齐。将 CPU 发出的内存指令以 64B 对齐之后，内存控制器会索引内存块落在按一个子树中，内存控制器能够得到对应的子树编号（假设我们子树的大小是 4M，那么 0-4M 的内存块都由编号为 0 的子树保护）。子树的编号可以进一步拆分为：tag+index+offset。tag 是用于区分在同一个缓存集合中的不同的缓存行，index 表示该子树是属于哪一个缓存集合的，offset 用于表示当前的子树属于一个缓存行中的第几个子树（因为一个缓存行中有 4 个子树

的根节点）。以上的方式和计算机中 cache 的组织结构非常相识，同时我们对子树缓存的驱逐策略采用了 LRU 的方式，即我们淘汰最老的为被访问到的缓存行。LRU 的策略能够通过增加一个比特位实现，内存控制器一旦访问了某个缓存行，就将对应的缓存行的 LRU 比特置为 1，当子树缓存行不够的时候，LRU 从上一次淘汰的缓存行开始，如果 LRU 比特是 1 那么就将它清零，如果 LRU 比特是 0 那么就驱逐该缓存行并且记录该位置。在子树缓存中保存了当前活跃的子树，缓存的大小决定了内存控制器中能够保存的活跃子树的大小，在该系统中，内存控制器中保存了 32 颗子树，能够支持 128M 的内存保护，这个大小和 SGX 中保护的内存的大小一致。在云场景中，内存控制器中保护的子树根节点个数可以更多，一支持更多的活跃子树，减小运行时候的开销。

Roor-of-root Root-of-root 是根数的根节点的哈希/计数值，不同于子树缓存：Mount Table，Root-of-root 中的数值常驻在内存控制器中，但是与 Mount table 不同的是，Root-of-root 可以非常小，往往只有几个 bit。Mount table 中空间不够的话的子树根节点会采用 LRU 的淘汰策略，将子树根节点淘汰到内存中。Root-of-root 只存在一个，并且在内存中并不存在备份。所以该值被固定在内存控制器中，以防止被恶意的攻击者篡改，Root-of-root 是整个完整性检查系统的可信基础。

内存控制器中新增加三个部件：Secure Bitmap；Mount Table；Root-of-root，同时在内存控制器中还需增加完整性保护的逻辑以及挂载子树的功能，这些将在之后的章节中详细的介绍。

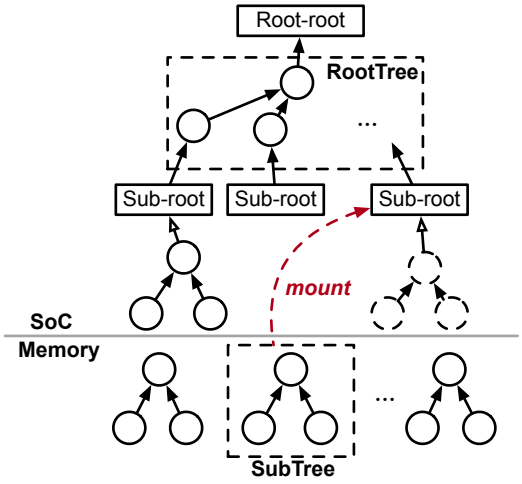

图 3–2 哈希森林

3.3 完整性保护树的动态原语

之前所有的工作中，完整性保护树都是静态的，不同于其他的树的组织结构，因为可以动态的增加一个分支，或者合并两个分支。完整性保护树不能够更改树的组织结构，因为内存控制器需要在获得内存地址的时候就知道其对应的每一层节点的在内存中具体位置，静态的树的组织结构能够不改变树中的每一层的结构以及对应所在的位置中，所以树中节点的位置在初始化的时候就能够确定了，方便内存控制器之后的读取与验证。静态的树在实现的时候虽然很简单，但是存在诸多弊端，比如只能够保护连续的内存空间，没有办法动态的更新保护的内存范围。所以我们希望给静态的完整性保护树附上一些动态操作的原语，

能够实现动态的内存你保护，同时让内存控制器能够快速的知道树中每个节点在内存中的位置。结合在上诉的活跃子树与不活跃子树的描述与分析，我们为 Mountable merkle tree 和哈希森林提出四个动态操作原语：挂载（Mount），卸载（Unmount），添加（add）和移除（remove）。通过这四个原语，我们可以加子树在活跃，不活跃与未分配三种状态间转换。如图 3–2所示，挂载（Mount）与卸载（Unmount）分别对应将子树的状态从活跃转换为不活跃以及从不活跃转换为活跃。挂载与卸载的想法类似与在操作系统中的文件系统中的挂载文件到文件树中。在 Mountable merkle tree 中，我们可以将一颗子树挂载到 Mount Table 中，在 Mount Table 中的子树的根节点以及子树的地址，能够被内存控制器直接读到，从而进行完整性保护的计算。而卸载的操作可以将一个挂载在 Mount Table 中的子树卸载到内存中，这样可以奖 Mount Table 中的空间留给其他的活跃的子树使用。添加与移除是针对分配与未分配子树而言的。添加操作是将子树添加在哈希森林中，同时奖 secure bitmap 中对应的子树的比特为设置成为 1，一颗子树添加到哈希森林中并不意味着他是活跃，而是处于不活跃的状态。同样我们也可以使用移除（remove）操作将一颗子树从哈希森林中移除，这个操作意味的该子树对应的内存不需要经过完整性保护。通过这四个完整性保护树的动态操作原语。我们可以实现动态的增加需要保护的内存，把活跃的子树挂载到 Mount Table 中来减少运行时候的开销。同时在添加一颗子树到哈希森林中的时候，我们需要指定该子树在内存中的位置，并且和子树初始化的哈希/计数值保存在内存中的一块特殊的区域：MMT 元数据区域中，之后在挂载的时候，会将该子树的地址写入 Mount Table 中，这样内存控制器就可以知道该子树的其他节点在内存中的位置，不依赖与任何运行时的信息。

3.4 内存布局

如图 3–1所示，为了适应动态可扩展的内存完整性保护，我们对内存中的布局做了进一步的细分。内存中的居于被分为四个部分，分别是：非安全内存，安全内存，子树节点区域以及 MMT 元数据区域。其中非安全内存不需要经过完整性验证，其对应的子树属于未分配的状态；安全内存是指该内存受完整性保护，它对应的子树处于已分配状态，但是不能保证子树一定处在活跃的状态；子树节点区域中存放着子树的中间节点，子树的根节点存在内存控制器中，但是因为内存中的空间是有限，不可能存下子树中的所有的节点，所以子树中的中间节点需要存放在内存中。子树节点区域可以动态的分配，往往实在动态添加安全内存的时候随之一起指定对应的子树节点区域。安全内存以及子树节点区域可以离散的分布在内存中，不一定需要时连续的内存区间。MMT 元素据区域（MMT metadata zone）是一块特殊的区域，该区域只能由拥有特权级的程序访问以及修改，MMR 元素剧区域中存储着所有的子树的根节点（包括了未分配即已分配的子树），根节点中包括了对应子树的哈希/计数值以及子树的在内存中地址，如果当前的子树没有分配，那么子树的根节点中的哈希以及地址为空。MMT 元素据区域中的另一部分时根数的节点。根数用来保护 MMT 元素据区域中所有子树的根节点的完整性。如果恶意的攻击者通过物理攻击方式修改 MMT 元素据区域中的数据将会被根数的检查发现，如果攻击者通过软件攻击的方式修改 MMT 元素据区域中的值将会被禁止，只有特权程序才能访问该区域，其他程序想要修改将会被禁止。下面我们将具体的介绍一下在 RISCV 架构中如何保护 MMT 元素据区域的

监视者 (Monitor): 在 RISCV 的指令集架构中分为四个特权级，分别时用户态（U mode），内核态（S mode），虚拟化层（H mode），监视器层（M mode）。其中 U mode 中运行着一般的用户程序，S mode 中运行着操作系统，H mode 中运行着虚拟机管理者（VMM），M mode

中运行的监视器。其中 M mode 拥有最高的特权级。M mode 可以访问所有的物理资源以及拥有自己的特权指令。M mode 中的代码非常少，一般情况下会把对物理资源的管理交给更低权限的程序例如运行在 S mode 中内核程序，或者运行在 H mode 中的 VMM。正常的 M mode 中值运行了启动机器的 bootloader 用于加载代码，以及少量的初实话工作。当然我们可以在 M mode 中加入一个监视着（monitor）来监管部分物理资源或者对其作检查，来提高系统的安全性。在 RISCV 的指令集架构中，存在着一些特殊的寄存器：pmp。pmp 寄存器时成对出现的，包括一个 pmp 地址寄存器以及一个 pmp 配置寄存，通过这两个寄存器，可以划初内存中的一块连续的内存区域，同时给该内存区域设置上访问的权限，在 RISCV 指令集中 pmp 寄存一般由 16 个，既可以划分出 16 个连续独立的内存区域。我们利用 M mode 中的监视器以及 pmp 寄存器来保护 MMT 元素据区域。首先我们选定一个 pmp 寄存器，然后给它分配一块物理内存作为 MMT 元素据区域。并且给该内存设置的访问修改的权限为 M mode，即只有处在 M mode 中的程序才能够访问 MMT 元素据区域中的信息。其他的特权级的程序如果访问该内存区域将会下陷到 M mode 中进行处理。通过这样的机制能够保证只有 M mode 中改的监视者能够修改 MMT 元素据区域中的数据其他任何的程序都无法修改其中的数据。从而保证了 MMT 元数据区域不会受到软件攻击。

3.5 完整性检查流程

下面将详细介绍如何进行完整性检查：

1. CPU 向内存控制器发送一个内存访问请求的包，内存控制器解析该内存访问的地址属于哪一个子树，然后再 secure bitmap 中检查该子树是否被分配，该内存访问是否落在安全内存中。如果落在安全内存中着进入步骤二，否者直接进入步骤六。
2. 内存控制器检查访问地址对应的子树是否再 Mount Table 中（通过检索 index 以及匹配 tag），如果子树再 Mount Table 者进入步骤五，否者进入步骤三
3. 子树不在 Mount Table 中，所以内存控制器根据 LRU 的方式选择一个合适的子树缓存行，将其中的子树的根节点淘汰到内存中 MMT 元素据区域中（一个子树缓存行中存储了四个连续的子树的根节点）。因为被淘汰到内存中的子树根节点中的数据可能发生变化，所以会更新根书中节点的数值，如果必要的话会更新内存控制器中的 Root-of-root 中的数值常驻在内存控制器中，但是与 Mount。
4. 内存控制器选择目标地址对应的子树，将子树挂载到 Mount table 中。再读取相应的子树的时候，需要将依据根数中节点哈希/计数值对 MMT 元数据区域中的子树根节点数据进行完整性验证，如果验证通过才能够将子树根节点加载到 Mount Table 中，因为一旦加载之后，对子树的访问将不会触发对根数的检查。
5. 现在对应子树的根节点已经在 Mount Table 中了，我们根据子树根节点中的哈希/计数值和子树的物理地址，找到对应的子树节点区域，然后通过子树根节点中改的哈希值，对访问内存中的数据进行完整性检查。如果检查通过则进入步骤六
6. 内存控制器对访问的内存进行读取操作，如果是读操作，那么将该内存块中的数据以及读出来然后用包的形式返回给 CPU，如果是写操作，那么将对应的数据写道内存块中，更新内存中的数值。

以上的步骤是 MMT 中为了自动实现挂载以及卸载操作所增加的内存完整性检查流程。下面会具体阐述如果通过 Mountable merkle tree 进行完整性检查，这个部分在上述步骤中的三四五中使用到，首先我们会具体介绍 Mountable merkle tree 中节点的组织结构。

3.5.1 树节点结构：三级 counter

下面我们会详细的介绍 Mountable merkle tree 中节点的组织形式。一个节点的大小为 512 比特，符合一条缓存行的大小，这一点是为了方便能够把整个节点缓存在系统的 cache 中减少读内存的操作。一个树节点的 512bit 被分成了五个部分，分别是：global counter，extra counter，index，local counter 以及 hash。首先将介绍 global conuter 以及 local counter 的作用。首先，为了提高节点中的扇出，我们用 counter 替代了 hash，这个做法在 BMT 中得到的使用，用 counter 代替 hash 能够很好兼顾加密以及完整性的保护，我为了减少受到 replay 攻击的可能，每一次对内存的写访问都会增加对应的 counter 的值。采用 global counter 和 local counter 的方式是在 VAULT 中首次提出在完整性保护中的。local counter 决定了一颗树节点的扇出的度数，而 global counter 决定了 counter 的大小（counter 的大小是防止重放攻击的关键）。每一个树节点的 counter 是由 local counter 和 global counter 拼接而成，所有的 local counter 公用一个 glocal counter，一旦 local counter 耗尽了，需要增加 global counter 数值时，就会使所有节点中的 counter 的数值均发生了变化，进而所有被 counter 保护的子节点都需要重新计算对应的哈希。重新计算哈希将带来不可忽略的开销，所以我们尝试优化树节点中的数据结构来缓减这部分的性能开销。

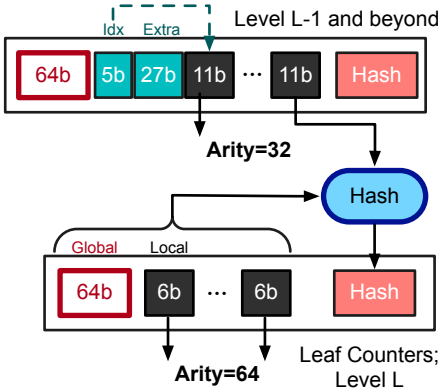

图 3–3 **MMT** 树节点数据结构：三级 counter

同 Mountable merkle tree 的想法类似，我们发现，在完整性保护树中，也存在一定的局域性，树中的节点被分成了热节点以及冷节点，其中热节点是快速增长的 counter 节点，冷节点是慢速增长的 counter 节点，因为树的组织形式，一个父节点将会对应多个子节点，即父节点保护的内存范围比子节点要大，在树的顶部的节点中，往往只会存在一个活跃的节点，又因为树中顶部的节点被访问的概率远超过底部的节点，所以 counter 会快速的增长，更加容易发生 overflow，所以我们区分对待同一个树节点中的不同的 counter。在一个树节点中可能存在一个或者少数几个快速增长的 counter 节点，剩下的 counter 的节点是不活跃的节点，增长的速率比较慢，根据以上的观察，我们除了将 counter 分成 global counter 和 local counter 之外，我们还加入了 *extra counter*，作为对热 counter 节点的补充。如图 3–3所示，我们在树节点中引入两个新的结构：**extra counter** 以及 **index**，其中 index 表示了当前的热 counter 节点是哪一个。如果一个 counter 节点是热节点，那么会使用 extra counter 中的值作为补充，如果当前的 counter 是冷节点，那么默认的 extra counter 为 0。在三层 counter 的结构下，当 local counter 发生了 overflow，首先回去检查该树节点中是否还存在空闲的 extra counter 可以使用，如果存在那么将 index 设置为对应的 counter，然后将 extra counter 加一。

如果当前的树节点中没有空闲的 extra counter 可以使用，我们将 global counter 加一，然后对所有的子节点重新计算哈希值，并将 index 设置为发生 overflow 的 counter 节点。如果 extra counter 发生了 overflow，那么也需要将 global counter 加一，然后对所有的子节点重新计算哈希。所以在三层 counter 的情况下，哈希的重计算可能发生在两种情况下：

1. 没有空闲的 extra counter 了
2. extra counter 发生了 overflow

虽然发生重新计算子节点哈希的情况变得复杂了，但是因为加入了 extra counter，可以充分利用 counter 的不均衡性，进一步减少 local counter 的大小，增加树节点的扇出，保护更多的内存。在 MMT 中树节点的扇出分别为：64，32，32，...，因为越往上的节点中的 counter 越容易被访问到，所以对应的扇出会减少，但是由于加入了 extra counter，local counter 的大小能够比 VAULT 更小（VAULT 中树节点的扇出分别是 64，32，24），相同层数下树的扇出更多，同时能够保护得内存也更大。值得注意的是，index 和 extra counter 的个数可以为多个。在扇出为 32 的树节点中：local counter 为 11 bit，global counter 为 64 bit，index 为 2×5 bit，extra counter 为 2×11 bit（在该配置下存在两个 index 和两个 extra counter）以及 hash 占用 64 bit，这五个部分相加刚好等于一条缓存行的大小 512 bit。

除了三层 counter 的设计，我们对节点的哈希计算采用了已有的方式，我们对父节点中对应的 counter 数值和子节点中的所有 counter 用 PMAC 算法进行哈希的计算（初始化的时候会随机生成用于 PMAC 计算的密钥），将得到的哈希值放在子节点中最后 64 bit 中。当发生一次写请求的时候，需要更新对应的 counter 数值，节点的哈希值会进行重计算，当发生一次读请求的时候，内存控制器会验证每一层树节点中的哈希值和计算出来的哈希值是否匹配，如果匹配才允许内存的访问。以上的步骤保证了 MMT 中的 counter 的完整性。对于任意一个内存数据块，有对应的 counter 和 MAC，其中 counter 受 MMT 中节点中的哈希保护，而内存中的数据受到 counter 以及对应的 MAC 保护。值得注意的是，对于 counter 和 MAC 我们只需要保证其中一个的完整性，就能够保证内存块中的数据的完整性。下面我们将介绍 PMAC 的算法：

3.5.2 PMAC

PMAC 全称 Parallel Message Authentication Code，是一种高效的验证数据完整的算法，它能够以对数据块进行并发的验证，在介绍 PMAC 之前我们将介绍 AES 加密算法。

AES 加密, AES 加密算法是一种高效的加密算法，器核心使用了异或操作，使得加密的过程能够在一个 cycle 内完成。AES 加密的核心公式是: 密文 = 明文 $\oplus$ pad，其中 pad 由因子（CTR）加密后生成，每一次加密的 pad 应该保持不同，因为攻击者可以通过明文和密文快速的得到 pad 的值（异或具有可逆性）。一般在生成 pad 的时候往往考虑计入一些时空因素，比如在完整性保护中，pad 是由 key，counter 以及对应的内存地址通过 AES128 等算法生成的，其中 counter 和地址作为 CTR。因为每一次的读写都会得 counter 加 1，对应不同内存操作其对应的内存地址也不一样，所以不可能同时存在两个相同的 pad，除非 counter 耗尽。我们得到 pad 之后，将明文和 pad 做一次异或的操作就能够得到对应的密文。异或操作在 cpu 中只需要一个 cycle 既可以完成，所以 AES 加密运算是一种高效的单次加密方式。（这里所说的 AES 加密不等价与 AES128 等算法，AES128 是用于生成 pad 的算法，生成 pad 之后还需要和明文做一次异或得到密文）

如图 3–4所示，在 PMAC 中，会将数据块划分成为等长 M 块，其中的每一块数据的长

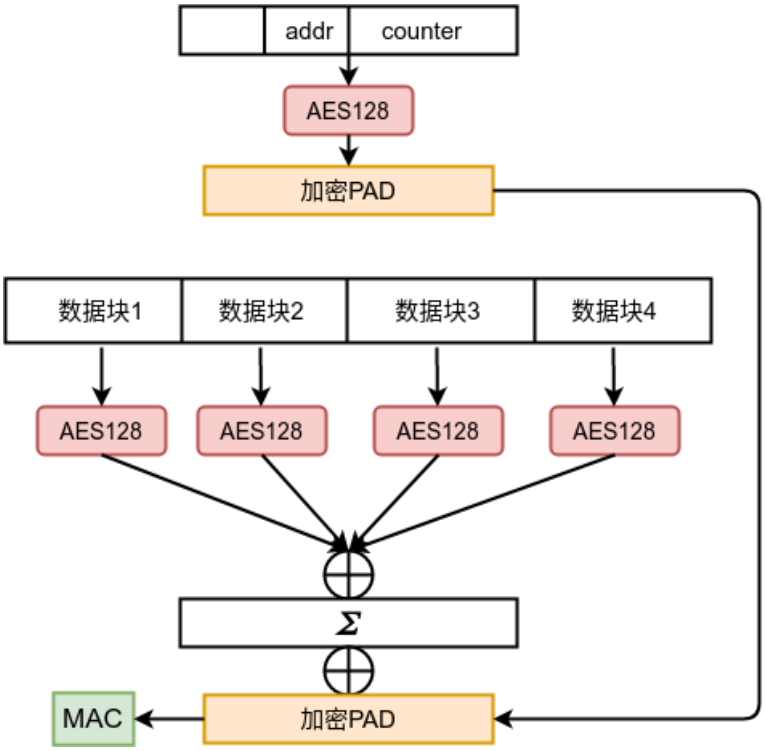

图 3–4 **PMAC**：并行计算数据块 MAC

度都是，e.g., 128 bit，同时 PMAC 还生成 key1，key2 用于 AES 加密计算。对分割好的 M 块数据分别用 AES(key1) 做加密，将加密后得到的 M 块密文块通过异或的方式生成 Σ。最后用 AES(key2，CTR) 得到对应的 pad（其中 CTR 为内存的 counter 和地址），然后对之前得到的 Σ 做异或运算，异或后的结果就是 PMAC 算法最终得到的 MAC 值。在 PMAC 的整个过程中，可以对分割好的 M 块数据同时进行 AES 加密，相较于 CMAC, HMAC 这类串行的 MAC 来说非常的高效，并且 AES 加密计算本身非常高效，所以 PMAC 被认为是一种高效的计算 MAC 函数。

3.6 启动阶段

在内存分布中，我们将内存分成了不可信的内存区域，安全的内存区域，子树节点区域以及 MMT 元素据据区域。其中只有 MMT 元数据区域需要在硬件启动的时候就被配置好，剩下的区域能够在之后被动态的分配。在安全启动中，通电之后，在硬件固件中的 bootloader 会首先被加载运行，首先 bootloader 会随机生成用于完整性检查的 PMAC 的两个密钥 key1，key2 同时 bootloader 会根据拥有的内存大小，指定一块区域为 MMT 元素据区域，并且用 PMP 将该区域保护起来，只有 M mode 程序能够访问该区域。之后 bootloader 会加载 monitor 的代码。因为 monitor 的代码需要完整性保护，所以 bootloader 需要将 monitor 所在的内存区域设置为安全内存区域，同时为 monitor 分配对应的子树的节点空间，将初始化的子树根节点哈希和地址填到对应的 MMT 元数据区域中，更新根数中的哈希值。之后将 monitor 的代码加载到安全内存中，同时更新对应的子树的哈希值。bootloader 加载完 monitor 的代码后，计算 monitor 中的代码的哈希，用于之后的验证。然后将控制权转交给 monitor，由 monitor 进行之后的启动代码。因此在 bootloader 启动的时候内存中的布局为：由 bootloader 中配置

到的 MMT 元数据区域，由 bootloader 分配的 monitor 的安全内存区域以及对应的子树节点区域。在 MMT 元数据区域中分配第一个子树用于保护 monitor 中的内存。当 monitor 启动之后，会按照需求，分配剩下的安全内存以及对应的子树，然后把对应的子树的根节点添加到 MMT 元数据区域中。整个安全启动的可信链为可信的 bootloader（硬件的固件中，通常不能够更新或者只能更新厂家提供的 bootloader），bootloader 配置 MMT 元数据区域以及 monitor 的安全内存，同时验证了 monitor 的代码的完整性。这两个操作保证了 monitor 的运行在安全的内存中并且 monitor 代码本身的完整性得到保证，然后运行 monitor，monitor 的安全 bootloader 保证，所以 monitor 被认为时安全，由 monitor 进行之后的可信内存的配置。

3.7 分配子树

不是所有的安全内存都需要在启动的时候配置，这也是 MMT 的一个特色即动态的分配安全内存。所以 monitor 在启动之后会负责分配安全内存以及对应的子树的空间，并且将初始化的子树根节点添加到 MMT 元数据区域中。为了保证分配子树的安全。monitor 对外暴露特定的 monitor call 的接口。非 M mode 的程序只能够通过调用给定的接口在可以配置安全的内存。*SBI_ALLOC_SECURE_MEMORY(addr, len)* 该接口用于将 [addr, addr+len] 的内存设置为需要安全内存，如果该内存区域已经为安全内存，那么将忽略之后的步骤 monitor 在分配的过程会从自己的内存中或者向 kernel 中索要一块内存用于存放该安全内存的子树节点。monitor 可以按照需将将安全内粗清零或者根据已有的数据计算出当前的子树中各节点的哈希值，然后将子树的根节点的哈希值以及对应的子树节点的地址填入 MMT 元数据区域中。然后 monitor 通过一条特殊的指令将内存控制器中的 secure bitmap 中对应的子树设置为 1，代表该子树处于已分配的状态。*SBI_FREE_SECURE_MEMORY(addr, len)*，首先 monitor 会通过内存控制器中的 secure bitmap 检查当前的内存区域是否时安全内存区域，如果当前内存不是安全内存那么将忽略之后的步骤。monitor 会清楚该内存在 seucre bitmap 的中的标志。如果内存对应的子树在 Mount table 中，则会将当前该子树的根节点驱逐初 Mount table 中。同时会清空 MMT 元数据区域中的对应的子树的根节点的哈希值和地址，同时清除子树节点中改的数据，防止信息泄露。通过这两个特殊的 SBI call 我们可以实现 mountable merkle tree 中的两个动态原语：添加（add）和移除（remove）。通过这两个动态原语我们可以实现动态的安全内存分配以及离散的内存保护。不需要再初始化的时候，就将所有的树的节点的内存空间分配好。树的节点所占用的内存空间为保护数据的大小的 15-25%，在云场景下，这部分的内存开销不可以忽略，特别时当需要保护的内存较少，非安全内存的较多的情况，子树节点的内存未被使用，但是也不能作为正常的非安全内存共程序使用。所以 MMT 提供了一种更加灵活的内存保护方案，尽量的减少额外的内存开销，提供了较高的可扩展性。

3.8 细粒度保护

之前我们所讨论的对内存的完整性保护的粒度都是以子树的粒度进行，但是子树的大小在不同的实现中并不一致（在本论文中子树的大小未 4M），这给上层的操作系统带来内存管理的不便。同时子树的粒度与页粒度相比较大，不利于细粒度的内存管理，所以我们希望能够有一种更加细粒度的完全内存的划分方式。Mountable merklr tree 可以支持更加细粒度的内存保护（最小为缓存行级 64B）。为了实现更细粒度的内存保护，需要软硬间结合实现，而不能仅仅靠内存控制器实现。在软件方面 monitor 可以通过 tag memory 或者控制页表等方式实现也页粒度的内存隔离，同时赋予 CPU 两个模式：安全模式与非安全模式。我们

认为在安全模式下，CPU 访问的内存都是安全内存，在非安全模式下，CPU 访问的内粗都是非安全内存。这样在内存控制器中就不需要关心访问的内存是否安全。但是由于挂载的单位是子树，如果子树的粒度为页大小，会导致子树过多，频繁的发生挂载与卸载的操作。如果不改变子树的大小，那么在一棵子树中，必然同时存在安全内存以及非安全内存。对于安全内存的访问，我们还是按照上面所说的完整性检查的流程进行。对于子树中的非安全内存，我们将其对应的 counter 全部设置为一个常量，对该内存的读写都不会去修改对应的 counter 的数值，而对安全内存的读写中会将非安全内存的 counter 当作一个常量加载进来，作为 PMAC 输入参数的一部分，进行完整性计算与验证。因为非安全内存对应的 counter 不会发生改变，所以并不影响安全内存的检验。当我们希望将非安全内存转换为安全内存的时候，我们可以将之前设置的常量 counter 作为初始值，结合非安全内存中的初始数据计算对应的哈希值，然后更新子树的根节点中的哈希值。通过这样的方式能够将非安全内存转换为安全内存，同时并不影响原有的安全内存的检查。

3.9 总结

在本论文中，我们提出了可挂载完整性保护树，哈希森林，子树与根数等概念，并且提出了四个完整性保护树的动态原语：挂载（Mount），卸载（Unmount），添加（add）与移除（remove）。另外通过观察我们发现了内存访问的存在局域性，冷热不平衡的现象，通过设计 Mount Table，三层 counter 等结构来解决内存访问不平衡的问题。我们在 RISCV 架构上实现了 MMT 的原型，它能够实现了对离散内存的完整性保护，支持动态的分配安全内存，同时将支持 TB 级别的内存完整性保护。可挂载完整性保护树能够兼容已有的完整性保护的数据结构，可以很方便的移植到现有的方案中。同时只需要对内存控制器做一定的修改，对 CPU 核心的修改较小，能够适应不同的指令集架构。可挂载完整性保护树的提出能够适应多种新型的应用场景，包括了具有高安全需求的可行执行环境 TEE，以及新型的内存存储 NVM 中。为解决内存完整性保护提供了新的支持

第四章　实现与测试

4.1　实现平台与环境

我们在 *gem5* 模拟器上实现了 MMT 的原型，*gem5* 是一个多指令集的全系统模拟器，它可以模拟不同的指令集的（X86, ARM, RISCV，MIPS, SPARC 等），不同的 CPU 模型：单周期，多周期，顺序执行，乱序执行，多发射等，不同的内存层次架构：无缓存，L1 缓存，L2 缓存，L2 缓存，以及指定每一级缓存的大小，集合，几路缓存等配置参数。同时 gem5 还对内存做了全模拟，实现了经典的内存模型以及 ruby 的内存模型。在模拟性能方面，gem5 可以使用精确时间的模拟 timing，原子的访问模拟 atomic，以及性能的访问模式 functional，*gem5* 为全系统模拟提供了诸多搭配的选择，同时也提供了较为精确的模拟性能。相较于其他的相关工作使用的的 trace 模拟器（USIMM），只能够根据内存的访问 trace 文件，来进行模拟执行，这样的的模拟的精确性将不如全系统模拟。同时全系统模拟可能准确的执行程序并且给出相应的计算结果，而不仅仅只是对性能的模拟。我们可以在 *gem5* 上运行的完整的 linux 系统，或者未经过修改的原程序。

表 4–1 **gem5** 中参数配置**.**

处理器	
指令集	RISCV
核数	4 核
频率	1GZ
L1d 缓存	两路，64K
L1i 缓存	两路，32K
L2 缓存	八路，2M
L3 缓存	十六路，16M
内存	
内存型号与频率	lddr3，800mHz
读队列	32
写队列	64
行缓存	1K
内存核心数	8
单通道 rank 数	2
单通道 bank 数	8
内存时序参数	
Tck	1.25ns
Tbust	5ns
Trcd	13,75ns
Tcl	13,75ns
Tras	35ns
Tcs	2.5ns

如表格 4–2所示，我们列举 gem5 配置的处理器，内存以及内存时序参数等。我们选择了指令集为 RISCV 的处理器，因为 RISCV 为开源指令集项目，可以方便做指令的修改以及后续的开发。同时市面上也存在诸多 RISCV 架构的处理和 FPGA，能较为方便的从模拟器中移植到真实的硬件中。我们采用了四核顺序处理的 CPU 核心，采用了三级缓存结构，其中 L3 缓存是共享的缓存。在内存控制器方面，内存控制器中的读队列和写队列分别为 32 和 64。我们模拟了 lddr3 的内存，800mHz 的频率，能够适应绝大多数的应用场景。其中单

内存条（单通道）上有两个 rank，每个 rank 上拥有 8 个内存核心。其余的内存中刷新频率 (Tck)，行地址到列地址延迟 (Tras) 等相关信息可以具体参考表 4–2

表 4–2 **MMT** 中参数配置.

MMT 配置	
子树层数	三层
子树保护内存	4M
根数层数	三层
子树保护内存	2M
MMT 元数据区域	≈2M
SoC 保护内存大小	128M
最大保护内存大小	512G
Mount Table	16K

4.2 测试

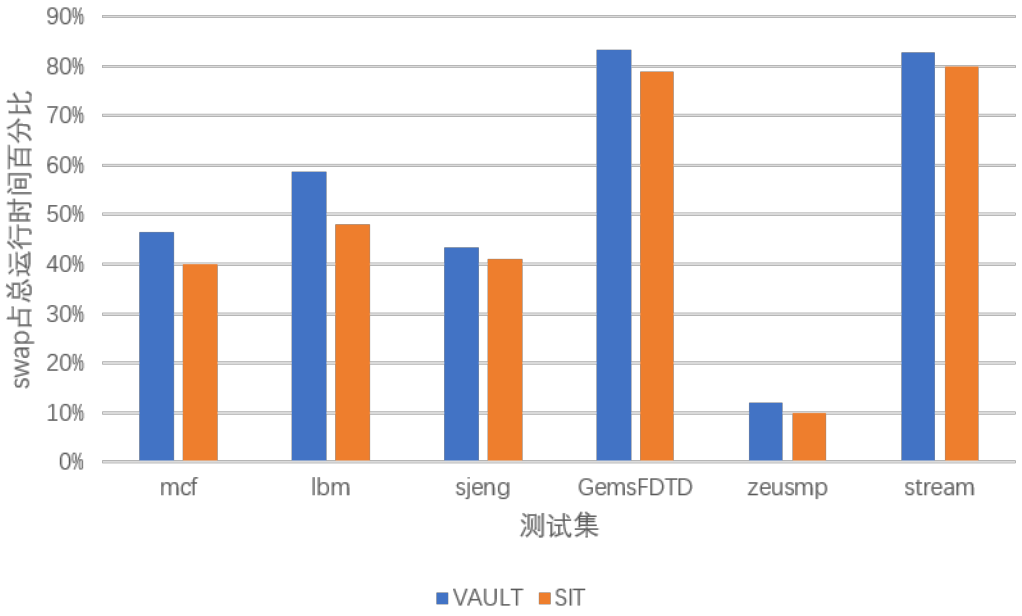

图 4–1 **SPECCPU, STREAM** 测试集在不同完整性保护方案下 **swap** 所占比列。

如图 4–1所示，我们分别测试了 SIT，VAULT 等先进的完整性保护方案，在超过 soc 上保护内存限制时候产生 swap 的开销占总运行时间的比例。为了能够理论上支持无限制的内存完整性保护方案，在 SIT 以及 VAULT 等方案中一旦使用的内存超过了 soc 可以保护的范围。CPU 会选择恰当页将其中改的数据进行加密，然后 swap 到非安全的内存中。为了保证从非安全内存中 swap 回到安全内存中，数据没有被修改。CPU 在 swap 的时候会生成一个 evict metatdate 结构。该结构中记录了 swap 出去页的哈希值，以及属于哪一个 enclave 等相关的元数据，该结构被存储在安全内存中。注意当安全内存十分紧张的时候，该数据结构也可能被 swap 到非安全内存中，同样我们会为存放 evict metadata 的页生成一个 evict metadata，类似于树的形式，只需要保存根上的 evict metadata 既可以保证所有 swap 出去页的完整性。虽然 SIT，vault 能够理论上支持更多的内存，但是当 SoC 上保护得内存耗尽的时候 swap 会

带来很严重的性能开销，测试发现，一次 swap 将会带来高达 40k 的开销，我们在 SPECCPU 和 STREAM 测试集上进行了测试（这里我们挑选了 SPECCPU 中运行时内存超过 128M 的测试用例），我们发现 swap 占据了总运行时间的 10%-80%，其中 5/6 的测试结果显示 swap 占据了总运行时间的 40% 以上，其中 GemsFDTD 和 STREAM 测试 swap 的开销格外严重，占据了总运行时间的 80% 以上，另外杜比 SIT 和 VAULT，在 VAULT 中 swap 的开销会更加的严重，这不是应为 VAULT 的设计的缺陷，还是因为 VAULT 运行时候的气态开销更小，导致了 swap 所占的时间更加明显。通过上面的数据显示，一旦使用的内存超过了 SoC 上保护的限制，swap 将造成验证的性能降级，所以优化 swap 的开销非常的重要。

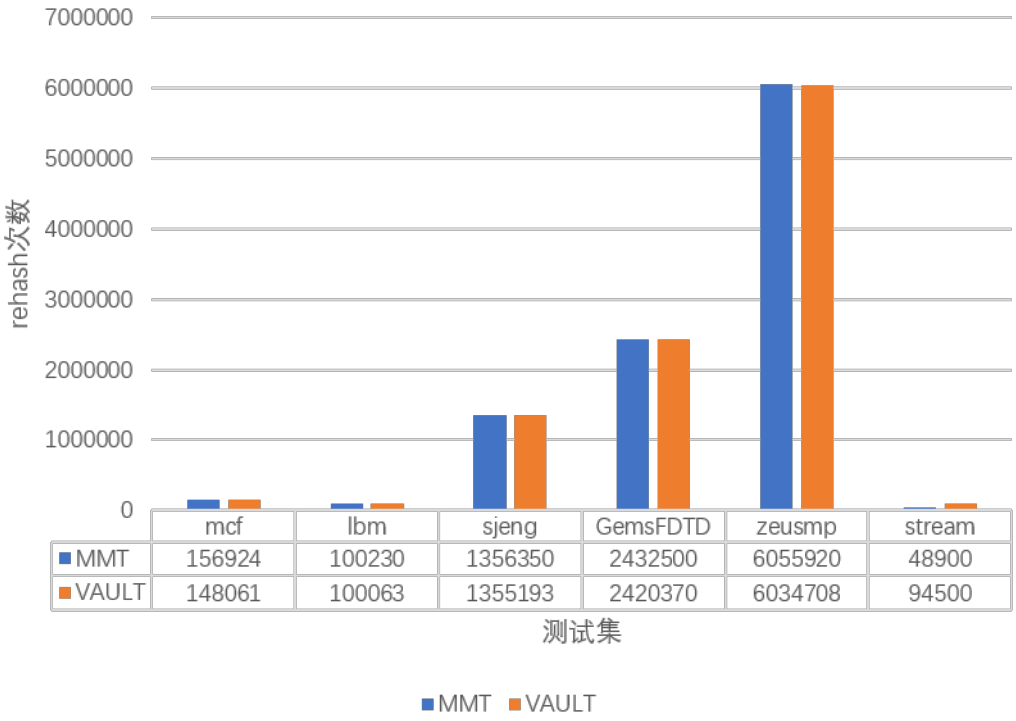

图 4–2 **SPECCPU, STREAM** 测试集在不同完整性保护方案下 **rehash** 的次数。

如图 4–2所示，我们测试了 VAULT 和 MMT 中 rehash 的次数。为了能够增加完整性保护树中节点的扇出，VAULT 和 MMT 都采用了分级 counter 的设计思路。因为分级 counter 会带来 rehash 的额外开销，来缓解 replay 等攻击。在 MMT 的设计中我们采用了三级 counter 的设计，三级 counter 的设计充分考虑了完整性保护树中的冷热 counter 不均衡，在较高的树节点中往往只存在一个活跃的 counter，而其他的 counter 可能处于不活跃的状态，所以我们可以减少冷 counter 比特数，增加热 counter 的比特数，从而兼顾安全性于节点的扇出。在这里我们同先进的 VAULT 完整性保护方案进行了比较。其中 VAULT 的稳定扇出为 24，而 MMT 中节点的扇出为 32。在更大的扇出情况下，MMT 并没造成更多的 rehash 的次数，说明在真实的 workload 下，完整性保护树中的冷热 counter 分布不均衡，而 extra counter 的设计能够很好的考虑到这种情况，从而进一步的提高树节点的扇出。在 SPECCPU 的测试集中，大多数的场景下，MMT 于 VAULT 的 rehash 的次数相差小于 1%，在 stream 的场景中，因为内存访问的比较规律，同时使用的内存较少，通过 extra conter 的方式能够使热 counter 拥有更多的比特数，从而减少热 counter rehash 带来的影响。所以相较于 VAULT 而言 rehash 的次数反而更少了。综上，我们认为 MMT 三层 counter 的设计充分考虑了完整性保护树的不

均衡性，在实际场景中不会带来更多的开销，在个别场景中，会比先经的 VAULT 的 rehash 开销更少。

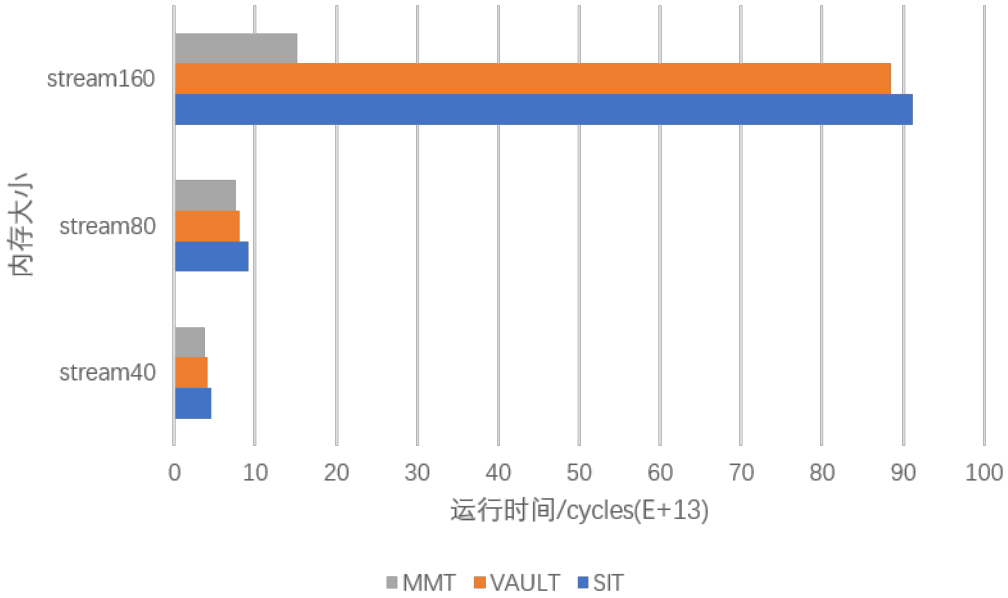

图 4–3 **STREAM** 测试集使用内存大小与不同完整性保护方案下性能关系图。

如图 4–3所示，我们比较程序使用不同的内存大小，对完整性检查的影响，在这里我们选取了 STREAM 测试集。STREAM 是为了测试内存的性能的测试集，拥有较大的内存压力，频繁的读写内存能够快速耗尽 SoC 上能保护得安全内存得空间，同时产生频繁得 swap 操作。这里我们分别选取了 SIT, VAULT, MMT 对使用 40M, 80M, 160M 内存得 STREAM 测试集做了分析。当程序使用的内存少于 SoC 上能够保护得内存时候，SIT，VAULT，MMT 之间得差距并不是很明显。SIT 会比 MMT 慢 20% 左右，和 VAULT 相比，MMT 采用了三级 counter 得设计，与 VAULT 相相比略有性能提升。因为 40M 和 80M 得内存较少，没有充分发挥三级 counter 得优势，当进一步扩大内存得时候三级 counter 得优势会更加的明显。当使用的程序使用的内存超过了 128M 得时候。SIT 和 VAULT 得性能开销急剧增加，因为 STREAM 会频繁循环的的访问内存，所以一旦超出了 SoC 上能够保护的内存时，就会频繁的发生 swap 操作，造成较大的性能开销。和 SIT，VAULT 相反，MMT 采用了挂挂载的方式代替内存的 swap，挂载操作的开销非常小（100 200 cycles）。所以当使用的内存扩大了一倍，运行时间也线性的扩大了一倍，挂载操作并没有成为性能的瓶颈。而 swap 操作会带来 7，8 倍的性能开销，说明 swap 操作取代了完整性保护的检查，成为了主要的瓶颈。

如图 4–4所示，我们在 SPECCPU 上测试了 SIT, VAULT, MMT, MMT+encrypt, 以及不做完整性检查原始的测试。其中 *gobmk*,*milc* 两个测试集使用的内存小于 128M，不会产生 swap 的开销，剩下的测试程序使用的内存从 [180M:1200M] 不等。从测试结果中我们可以发现，MMT 的性能均优于 SIT 和 VAULT，在 mcf 的测试中，SIT 和 VAULT 相较于没有完整性保护的程序分别会有 2.05x 和 1.6x 的开销，而 MMT 只有 0.4x 的开销，这部分的开销主要是完整性检查的开销，而挂载的开销小于 1%（挂载的操作只需要 100 200 的开销），在其他的测试中例如 GemsFDTD，SIT 和 VAULT 相较于不做完整性检查的程序来说有高达 7.5x 和 7.06x 倍的性能开销，而 MMT 只会有 0.35x 的性能的开销。在这种极限的场景下，MMT 相

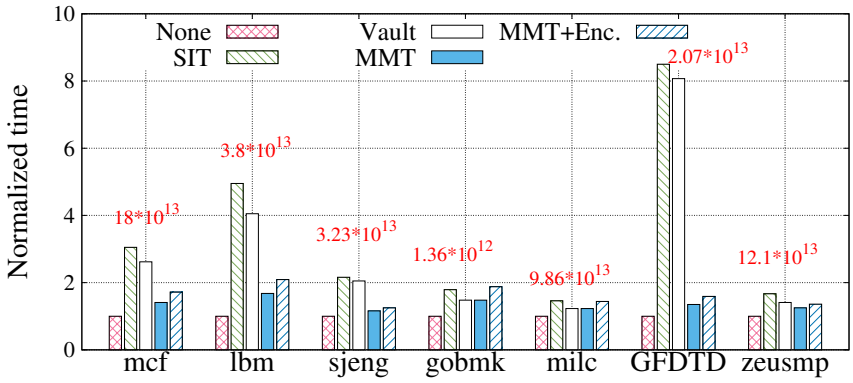

图 4–4 **SPECCPU** 测试集在不同完整性保护方式下的性能. *None* 表示没有完整性保护下的理论性能。

较于没有完整性保护的原程序来说只有 0.35x 的开销，Mount 操作大幅减少了原有的 swap 的开销，并且在绝大多数的场景下，Mount 的开销可以忽略不记（<1%）。当使用内存小于 SoC 上可以保护的内存时候，MMT 和 Vault 相比性能相近（这里为了是 SoC 上保护的内存相同，我们减少了 MMT 树的个数，因此三级 counter 的优势没有很好的体现）。根据上面的测试我们可以得出，不论在内存使用超过 SoC 能保护的内存大小，还是小于 SoC 保护的内存时候，MMT 都具有最好得性能，特别是当使用得内存超过 SoC 保护得内存大小得时候，挂载操作将极大得缓减原本 swap 页所带来的开销，又因为在运行时，可以通过 Mount Table 中读取子树的根节点来保证运行时候对完整性树检查的层数不变，从而保证 SoC 上内存足够的情况下，也不会带来额外的开销。

全文总结

参考文献

[1] MERKLE R C. A digital signature based on a conventional encryption function[C]//Conference on the theory and application of cryptographic techniques. [S.l. : s.n.], 1987: 369-378.

[2] DIFFIE W, HELLMAN M. New directions in cryptography[J]. IEEE transactions on Information Theory, 1976, 22(6): 644-654.

[3] ROGERS B, CHHABRA S, PRVULOVIC M, et al. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly[C]//40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007). [S.l. : s.n.], 2007: 183-196.

[4] TAASSORI M, SHAFIEE A, BALASUBRAMONIAN R. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures[C]//Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. [S.l. : s.n.], 2018: 665-678.

致　谢

感谢那位最先制作出博士学位论文 LaTeX 模板的交大物理系同学！
感谢 William Wang 同学对模板移植做出的巨大贡献！
感谢 @weijianwen 学长一直以来的开发和维护工作！
感谢 @sjtug 以及 @dyweb 对 0.9.5 之后版本的开发和维护工作！
感谢所有为模板贡献过代码的同学们，以及所有测试和使用模板的各位同学！
感谢 LaTeX 和 SJTUThesis，帮我节省了不少时间。

SCALABLE MEMORY INTEGRITY PROTECTION DESIGN AND RESEARCH

An imperial edict issued in 1896 by Emperor Guangxu, established Nanyang Public School in Shanghai. The normal school, school of foreign studies, middle school and a high school were established. Sheng Xuanhuai, the person responsible for proposing the idea to the emperor, became the first president and is regarded as the founder of the university.

During the 1930s, the university gained a reputation of nurturing top engineers. After the foundation of People's Republic, some faculties were transferred to other universities. A significant amount of its faculty were sent in 1956, by the national government, to Xi'an to help build up Xi'an Jiao Tong University in western China. Afterwards, the school was officially renamed Shanghai Jiao Tong University.

Since the reform and opening up policy in China, SJTU has taken the lead in management reform of institutions for higher education, regaining its vigor and vitality with an unprecedented momentum of growth. SJTU includes five beautiful campuses, Xuhui, Minhang, Luwan Qibao, and Fahua, taking up an area of about 3,225,833 m2. A number of disciplines have been advancing towards the top echelon internationally, and a batch of burgeoning branches of learning have taken an important position domestically.

Today SJTU has 31 schools (departments), 63 undergraduate programs, 250 masters-degree programs, 203 Ph.D. programs, 28 post-doctorate programs, and 11 state key laboratories and national engineering research centers.

SJTU boasts a large number of famous scientists and professors, including 35 academics of the Academy of Sciences and Academy of Engineering, 95 accredited professors and chair professors of the "Cheung Kong Scholars Program" and more than 2,000 professors and associate professors.

Its total enrollment of students amounts to 35,929, of which 1,564 are international students. There are 16,802 undergraduates, and 17,563 masters and Ph.D. candidates. After more than a century of operation, Jiao Tong University has inherited the old tradition of "high starting points, solid foundation, strict requirements and extensive practice." Students from SJTU have won top prizes in various competitions, including ACM International Collegiate Programming Contest, International Mathematical Contest in Modeling and Electronics Design Contests. Famous alumni include Jiang Zemin, Lu Dingyi, Ding Guangen, Wang Daohan, Qian Xuesen, Wu Wenjun, Zou Taofen, Mao Yisheng, Cai Er, Huang Yanpei, Shao Lizi, Wang An and many more. More than 200 of the academics of the Chinese Academy of Sciences and Chinese Academy of Engineering are alumni of Jiao Tong University.